TURBO VISION™ FOR C++

USER'S GUIDE

BORLAND

ソフトウェアのご使用条件

ユーザーの皆様へ

この度は、Borland International 社（以降ボーランド社）の製品をお買い上げいただき誠にありがとうございます。この製品は、ボーランド社の規定した『ナンセンスでないライセンス規定』およびボーランドジャパンの規定した『カスタマーサポート』に同意いただいた場合のみ提供させていただくことになっています。ご熟読のうえ、条件に同意された場合のみご開封ください。下記の規定をご熟読のうえ、添付されている「ボーランドジャパンカスタマーサポートセンター登録書」をボーランドジャパンあてにご返送ください。登録書をご返送いただいた方は、弊社でユーザー登録をおこないバージョンアップのご案内などを行います。登録書は、楷書体で丁寧にご記入ください。

登録書をご返送いただけない場合でも、フロッピーディスクの包装を解いた時点で、下記の規定にご同意いただけたものといたします。下記の規定にご同意いただけない場合は、ご購入されたパッケージを購入伝票などを添えて返品して下さい(ただし返送料金はお客様負担とさせていただきます)。フロッピーディスクの包装を解いた製品は、返品できません。

『ナンセンスでないライセンス規定』および制限付保証

ボーランドは、ボーランド言語製品のライブラリ・ソースコード（以降「ソース・プログラム」)の実施契約を行っております。ただし、8087 エミュレータ、オーバーレイ・メモリマネージャ、グラフィックスライブラリ（BGI）のソースプログラムは含みません。

1） 著作権

本ソフトウェアは米国著作権法、国際条約によって保護されております。

2） コピーの制限

本製品を使用するコンピュータに組み込むためのコピー、および破損および紛失などの損害から守るための保存コピー（バックアップ・コピー）のみ許諾します。

3） 使用権

書籍と同時にお考え下さい。一冊の書籍を 2 つの場所で、2 人が同時に読むことはできません。それと同時に、本ソフトウェアを 2 つの場所で、2 人が同時に使用することはできません。ただし、本ソフトウェアを同時でなければ複数の人が利用でき、複数のコンピュータの間を自由に移動することが可能です。(もちろん、ボーランドの著作権を侵害しないという条件つきです。)

ソース・プログラムは、ボーランド言語製品の実施許諾コピーをサポートするために使用できます。これは、ソース・プログラムを皆様自身が開発するボーランド言語ベースのプログラムのコピーに含めることができるということです。しかし、それは実行可能な形式においてのみ配布できるという意味です。実際のボーランド言語製品のライブラリー・ソースプログラムのいかなる部分も配布できません。も

ちろん、皆様自身のソースプログラムの配布は制限されません。ボーランド言語製品のソース・プログラムの一部または全部を譲渡、賃貸、販売することは認められません。

皆様はソース・プログラムを変更することはできますが、変更されたソース・プログラムの所有権は、その変更の程度に関わらず、所有権はボーランドに属します。皆様はボーランドの著作権、およびその他の財産権に関する表示を除去したり変更することはできません。また、変更の程度に関わらず、ソース・プログラムのいかなる部分も配布できず、他のコンピュータのオペレーティング・システムまたは環境に移すこともできません。皆様は、皆様自身によるソース・プログラムの変更、およびそのプログラムを含む製品に起因するすべての請求、責任および損害についての責任を負わなければなりません。

制限付保証

ボーランド・インターナショナル（以下「ボーランド」）は、同封するディスケット本体とドキュメンテーションが、購入後60日間は物理的不備や欠陥がないことを保証します。保証期間中に、材料または仕上げに欠陥が生じたことを書面を以て通知を受けると、ボーランドは、この通知が正しいことを確認し、欠陥ディスケットまたはドキュメンテーションを交換します。この制限付保証の違反に対する責任および救済処置は、欠陥ディスケットまたはドキュメンテーションの交換に限定され、ボーランドがこのような損害が生じる可能性について予め知らされている場合であっても、いかなる損害賠償の責任も負いません。損害には、これだけに限定されるものではありませんが、利益やデータの損失、ソース・プログラムの使用、特殊な損害、偶発的または必然的損害、またはその他の類似の請求が含まれます。いかなる場合においても、皆様またはその他の人々が受けた損害に対するボーランドの責任は、その賠償請求形式を問わず、ソース・プログラムを使用するライセンス料の表示価格または実際に支払われた価格を超えないものとします。

ボーランド・インターナショナルは、明示的または黙示的保証を含め、他のいかなる保証も拒否します。黙示的保証には、これだけに限定されるものではありませんが、市場性または具体的目的への適合性が含まれます。具体的には、ボーランドは、このソース・プログラムが特定の目的に適合することを主張または保証しておらず、市場性に対する黙示的保証は、ディスケットおよびドキュメンテーションのみを対象とした（ソース・プログラムは含まれません）限定保証の保証期間60日に限定され、それ以外については、明白かつ明瞭に拒否されます。

準拠法と一般規定

本ライセンスおよび制限付保証は、カリフォルニア州法に準拠して解釈、規制されるものとします。本パッケージ中のコンピュータ・ソフトウェアおよびドキュメンテーションを米国政府が使用、複写または公開する場合も、市販のコンピュータ・ソフトウェアにて起用される非公開権（DFARS 52227-7013）に準拠するものとします。本文書に明記されていない全ての権利は、ボーランドが保留します。

『カスタマーサポート』

アップデート、新製品情報、その他のご案内をさせていただきます。ただし、ライブラリ・ソースコードの使用に関し、技術的な援助は行われず、プログラミング技法に関する質問もお答えできません。

1） マスターディスクの再コピー（有料）

なんらかの原因でマスターディスクを破損された場合、再コピーサービスを受けることができます。破損されたディスクと再コピー実費 3,000 円（税別）を、弊社業務課までお送りください。

3） 仕様の変更

本ソフトウェアの仕様は、改良のため予告なく変更されることがあります。あらかじめご了承ください。

Turbo Vision™ for C++

ユーザーズガイド

Copyright © 1991
All Rights Reserved

Borland International
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95067-0001

編訳　ボーランド株式会社 1991

本マニュアルの一部または全部を，許可なく複製または転載することはできません。

Borland のすべての製品は，米国 Borland International, Inc. の米国における商標または登録商標です。他のブランドおよび製品名は，その版権所有者の商標または登録商標です。

目　次

はじめに …… 1
なぜ Turbo Vision なのか？ …… 2
Turbo Vision とは？ …… 2
知っておかなければならないこと …… 3
このマニュアルの内容 …… 3
Turbo Vision のインストール …… 4
INSTALL プログラムの使い方 …… 4
README ファイルと HELPME! ファイル …… 5
README ファイル …… 5
HELPME!.DOC ファイル …… 6
このマニュアルで使う書体 …… 7
製品に関するお問い合わせ …… 8

第1部　はじめての Turbo Vision …… 11

第1章　車輪の継承 …… 13
ウィンドウ型アプリケーションのフレームワーク …… 14
新しい視点から見たアプリケーション開発 …… 15
Turbo Vision のアプリケーションにおける各要素 …… 17
部品の名前付け …… 17
ビュー …… 17
イベント …… 18
ミュートオブジェクト …… 18
共通の“ルックアンドフィール”（外観と感触） …… 19
Turbo Vision 流“Hello, World!” …… 22
HELLO.CPP を動かしてみよう …… 23
メニューをプルダウンする …… 24
ダイアログボックス …… 25
ボタン …… 26

抜け出し方……27
HELLO.CPP の内部構造……28
アプリケーションクラス……28
ダイアログボックスオブジェクト……31
実行の流れとデバッグ……31
HELLO のメインプログラム……32
アプリケーションのインスタンス化……32
run メンバ関数……34
アプリケーションデストラクタ……34
まとめ……35

第2章　Turbo Vision アプリケーションの作成……37
最初の Turbo Vision アプリケーション……38
デスクトップ，メニューバー，ステータスライン……41
デスクトップ……43
ステータスライン……44
新しいコマンドの作成……47
メニューバー……48
Turbo Vision フレームワークについての補足……51
ウィンドウの操作……53
ウィンドウの構成……55
挿入メンバ関数……56
ウィンドウのクローズ……56
ウィンドウの動作……57
ウィンドウの内部……58
何が見える？……61
write の扱い方……62
簡単なファイルビュアー……63
テキストファイルの読み出し……63
表示のバッファリング……64
表示バッファ……64
テキストをバッファに移す……66
バッファの内容の表示……66
何行表示するかを知る……67
スクロールアップ/スクロールダウン……68

1個のウィンドウに複数のビューを持たせる……70
どこにどういう機能を持たせるか……73
ダイアログボックスの作成……74
モードつきダイアログボックスを実行する……76
コントロールの設置……77
ボタン，ボタン，...……78
ノーマルボタンとデフォルトボタン……80
フォーカスコントロール……80
好みに合わせた選択……81
クラスタの作成……81
チェックボックスの値……82
クラスタをもう1個……82
コントロールのラベル付け……83
入力行クラス……84
データの設定と取得……85
ホットキーとその衝突……89
その他のダイアログボックスコントロール……92
固定文字列……92
リストビュアー……92
リストボックス……93
ヒストリー……93
標準ダイアログボックス……94

第2部　Turbo Vision のプログラミング……95

第3章　クラス階層……97
階層の概要……98
階層作成について……101
クラスの分類……102
シードクラス……102
空のメンバ関数……103
クラスのインスタンス化と導出……104
インスタンス化……104
導出……105
メンバ関数……106

空のメンバ関数 ……106
デフォルトメンバ関数 ……106
仮想メンバ関数 ……106
仮想でないメンバ関数 ……107
静的メンバ関数 ……107
Turbo Vision のデータメンバ ……108
基本的なクラス ……110
TPoint ……110
TRect ……110
TObject ……111
ビュー ……112
ビューの概要 ……112
グループ ……112
TGroup クラス ……113
デスクトップ ……113
プログラム ……114
アプリケーション ……114
ウィンドウ ……114
ダイアログボックス ……115
ターミナルビュー ……115
フレーム ……115
ボタン ……115
クラスタ ……116
メニュー ……116
ヒストリー ……117
入力行 ……117
リストビュアー ……118
スクロールクラス ……118
テキストデバイス ……119
固定文字列 ……119
ステータスライン ……120
ストリーム ……121
コレクション ……122
ソートされたコレクション ……122
文字列コレクション ……123

リソース……123
リソースコレクション……123
文字列リスト……124

第4章　ビュー……125
“コントロールはテレビからもってきました”……126
単純なビューオブジェクト……127
まずは，基本的なオブジェクトについて……128
TPoint の取得……128
TRect の導入……129
Turbo Vision の座標……130
ビューの形態……131
守備範囲……131
要求に基づく表示……131
ふさわしい装いをする……132
複合ビュー……133
グループとサブビュー……133
グループの導入……134
別の角度から捉えた Z オーダー……135
グループの概要……137
ビュー間の関係……138
クラス階層……139
所有関係……139
サブビューとビューツリー……140
オーナーとサブビュー……140
選択されているビューとフォーカスビュー……144
フォーカスビューの検出……145
ビューがフォーカスを得るのはどんなときか？……145
フォーカスチェイン……146
モードつきビュー……147
デフォルト動作の変更……149
options フラグ……149
ofSelectable……150
ofTopSelect……150
ofFirstClick……150

ofFramed …… 151
ofPreProcess …… 151
ofPostProcess …… 151
ofBuffered …… 151
ofTileable …… 152
ofCenterX …… 153
ofCenterY …… 153
ofCentered …… 153
growMode フラグ …… 153
gfGrowLoX …… 153
gfGrowLoY …… 154
gfGrowHiX …… 154
gfGrowHiY …… 154
gfGrowAll …… 154
gfGrowRel …… 154
dragMode フラグバイト …… 155
dmDragMove …… 155
dmDragGrow …… 155
dmLimitLoX …… 155
dmLimitLoY …… 155
dmLimitHiX …… 156
dmLimitHiY …… 156
dmLimitAll …… 156
state フラグと setState …… 156
状態の変化に伴う動作 …… 157
ビューに何色を使うか？ …… 160
カラーパレット …… 160
カラーパレットの内部構造 …… 161
getColor メンバ関数 …… 162
デフォルトカラーのオーバーライド …… 163
新たに色を追加するには …… 164

第5章　イベント駆動型プログラミング …… 167
Turbo Vision に魂を入れる …… 168
ユーザ入力の読み込み …… 168

イベントの性質 ……171
　イベントの種類 ……171
　　マウスイベント ……172
　　キーボードイベント ……173
　　メッセージイベント ……173
　　“なし”イベント ……173
　イベントとコマンド ……173
イベントの運ばれる経路 ……175
　イベントの出所は？ ……175
　イベントの送り先はどこか？ ……176
　　ポジショナルイベント ……176
　　フォーカスイベント ……177
　　ブロードキャストイベント ……178
　　ユーザ定義のイベント ……179
　イベントのマスク ……179
　フェーズ ……180
　　フェーズデータメンバ ……181
コマンド ……184
　コマンドの定義 ……184
　コマンドのリンク ……185
　コマンドを有効/無効にする ……186
イベント処理 ……187
イベントオブジェクト ……189
　イベントのクリア ……191
　廃棄イベント ……191
イベントメカニズムの変更 ……193
　イベントの一括取り扱い ……193
　getEvent のオーバーライド ……194
　アイドルタイムの利用 ……195
ビュー間の通信 ……196
　仲介ビュー ……196
　ビュー同士のメッセージ ……197
　ブロードキャストは，どこで処理される？ ……198
　　誰か外に出てる？ ……199
　　最前面にいるのは誰？ ……200

handleEvent の呼び出し ……201
ヘルプコンテキスト ……201

第6章 信頼性の高いプログラムを開発するには ……203
全か無かのプログラミング ……204
セーフティプール ……204
従来の難しい方法 ……205
新しい簡単な方法 ……206
validView メンバ関数 ……207
delete と destroy ……207
メモリエラー以外のエラー ……208
エラーの報告 ……209
大量消費ビュー ……210

第7章 コレクション ……213
TCollection クラス ……215
動的なサイズ決定 ……216
コレクション要素の型の混在 ……216
コレクションの作成 ……217
繰り返し処理メンバ関数 ……220
forEach 繰り返し処理メンバ関数 ……220
繰り返し処理関数 firstThat, lastThat ……221
ソート済みコレクション ……224
文字列コレクション ……227
再度繰り返し処理関数について ……229
要素の検索 ……229
多態的なコレクション ……231
コレクションとメモリ管理 ……235
ヒープの利用可能性 ……236

第8章 ストリーム入出力可能なオブジェクト ……237
ストリームは無限 ……239
オーバーロード演算子 << および >> ……241
ストリーム入出力可能なクラスと TStreamable ……244
ストリームマネージャ ……245

ストリーム入出力可能なクラスのコンストラクタ ……248
ストリーム入出力可能なクラスの名前 ……250
ストリームマネージャの使用 ……251
ストリームマネージャのプログラムへのリンク ……251
ストリームオブジェクトの作成と使用 ……251
ストリーム上のコレクション ……253
write 関数の追加 ……253
デスクトップのセーブと復元 ……255

第9章　リソース ……257
リソースが必要なわけ ……260
リソースの内部 ……261
リソースの作成 ……262
リソースの読み出し ……265
文字列リスト ……267
文字列リストの作成 ……269

第10章　ヒントとコツ ……271
Turbo Vision アプリケーションのデバッグ ……271
実行されないプログラム ……271
イベントがマスクされている ……272
盗まれたイベント ……272
親のせい ……273
意図したとおりにならない場合 ……273
ハングアップした場合 ……274
Turbo Vision への移植 ……275
既存のプログラムの手直し ……275
仕組みを考え直して ……276
ビットマップフィールドの使い方 ……278
フラグ値 ……278
ビットマスク ……278
ビット演算 ……279
ビットのセット ……279
ビットクリア ……280
ビット反転 ……280

ビットテスト ……280
マスクの使用 ……281
まとめ ……281

第3部 Turbo Vision リファレンス ……283

第11章 リファレンスの使い方 ……285
必要な情報の見つけ方 ……286
オブジェクトの概要 ……287
名前の付け方 ……287

第12章 ヘッダファイルクロスリファレンス ……289
APP ヘッダファイル ……291
BUFFERS ヘッダファイル ……291
CONFIG ヘッダファイル ……292
DIALOGS ヘッダファイル ……292
DRAWBUF ヘッダファイル ……293
MENUS ヘッダファイル ……293
MSGBOX ヘッダファイル ……294
OBJECTS ヘッダファイル ……295
RESOURCE ヘッダファイル ……295
SYSTEM ヘッダファイル ……296
TEXTVIEW ヘッダファイル ……296
TKEYS ヘッダファイル ……297
TOBJSTRM ヘッダファイル ……300
TTYPES ヘッダファイル ……300
TV ヘッダファイル ……301
TVOBJS ヘッダファイル ……303
VIEWS ヘッダファイル ……303
クラス階層図 ……307

第13章 クラスリファレンス ……309
この章の見方 ……309
TSample class ……310
データメンバ ……310

メンバ関数 ……311
CharScanType ……311
fpbase ……312
メンバ関数 ……312
fpstream ……314
メンバ関数 ……314
ifpstream ……315
メンバ関数 ……315
iopstream ……316
メンバ関数 ……317
ipstream ……317
メンバ関数 ……317
フレンド ……319
KeyDownEvent ……320
MessageEvent ……320
ofpstream ……321
メンバ関数 ……321
Operators ……322
opstream ……323
メンバ関数 ……324
フレンド ……326
pstream ……327
データメンバ ……327
メンバ関数 ……328
フレンド ……330
TApplication ……330
メンバ関数 ……331
TAttrSelector ……332
メンバ関数 ……332
関連する関数 ……334
TBackground ……334
データメンバ ……334
メンバ関数 ……334
関連する関数 ……336
TBackgound のパレット ……336

TBufListEntry ……337
TButton ……337
　データメンバ ……338
　メンバ関数 ……339
　関連する関数 ……342
　TButton のパレット……342
TCheckBoxes ……343
　データメンバ ……343
　メンバ関数 ……344
　関連する関数 ……345
　パレット ……346
TCluster ……346
　データメンバ ……347
　メンバ関数 ……347
　関連する関数 ……351
　パレット ……351
TCollection……352
　データメンバ ……352
　メンバ関数 ……353
　関連する関数 ……354
TColorDialog……354
　データメンバ ……355
　メンバ関数 ……356
　関連する関数 ……358
TColorDisplay ……358
　データメンバ ……359
　メンバ関数 ……359
　関連する関数 ……360
TColorGroup ……360
　データメンバ ……361
　メンバ関数 ……361
TColorGroupList ……361
　データメンバ ……362
　メンバ関数 ……362
　関連する関数 ……363

TColorItem……364
データメンバ……364
メンバ関数……365
TColorItemList……365
データメンバ……366
メンバ関数……366
関連する関数……367
TColorSelector……367
データメンバ……368
メンバ関数……368
関連する関数……369
TCommandSet……369
メンバ関数……370
フレンド……371
TDeskInit……372
メンバ関数……372
TDeskTop……373
データメンバ……373
メンバ関数……374
関連する関数……376
TDialog……377
メンバ関数……377
関連する関数……379
パレット……379
TDisplay……380
TDrawBuffer……382
データメンバ……382
メンバ関数……383
フレンド……384
TEvent……385
TEventQueue……386
メンバ関数……386
フレンド……387
TFrame……388
メンバ関数……388

フレンド ……………………………………………………………………………390
関連する関数 ………………………………………………………………………390
パレット ……………………………………………………………………………390
TGroup ………………………………………………………………………………391
データメンバ ………………………………………………………………………392
メンバ関数 …………………………………………………………………………394
フレンド ……………………………………………………………………………405
関連する関数 ………………………………………………………………………405
THistInit………………………………………………………………………………406
メンバ関数 …………………………………………………………………………406
THistory………………………………………………………………………………407
データメンバ ………………………………………………………………………407
メンバ関数 …………………………………………………………………………408
関連する関数 ………………………………………………………………………410
パレット ……………………………………………………………………………410
THistoryViewer ……………………………………………………………………410
データメンバ ………………………………………………………………………411
メンバ関数 …………………………………………………………………………411
パレット ……………………………………………………………………………412
THistoryWindow……………………………………………………………………413
データメンバ ………………………………………………………………………413
メンバ関数 …………………………………………………………………………413
パレット ……………………………………………………………………………414
THWMouse …………………………………………………………………………415
データメンバ ………………………………………………………………………415
メンバ関数 …………………………………………………………………………415
TInputLine……………………………………………………………………………417
データメンバ ………………………………………………………………………418
メンバ関数 …………………………………………………………………………419
関連する関数 ………………………………………………………………………423
パレット ……………………………………………………………………………423
TLabel…………………………………………………………………………………423
データメンバ ………………………………………………………………………424
メンバ関数 …………………………………………………………………………424
関連する関数 ………………………………………………………………………426

パレット ……426
TListBox ……427
データメンバ ……428
メンバ関数 ……428
関連する関数 ……431
パレット ……431
TListViewer ……431
データメンバ ……432
メンバ関数 ……433
関連する関数 ……437
パレット ……438
TMenuBar ……438
メンバ関数 ……439
関連する関数 ……440
パレット ……440
TMenuBox ……441
メンバ関数 ……441
関連する関数 ……443
パレット ……443
TMenuView ……443
データメンバ ……444
メンバ関数 ……444
関連する関数 ……447
パレット ……448
TMouse ……449
メンバ関数 ……449
TMouseEventType ……451
TNSCollection ……451
データメンバ ……452
メンバ関数 ……453
TNSSortedCollection ……459
データメンバ ……460
メンバ関数 ……460
TObject ……462
メンバ関数 ……462

TPalette ……463
　メンバ関数 ……464
TParamText ……464
　データメンバ ……465
　メンバ関数 ……465
　関連する関数 ……466
　パレット ……467
TPoint ……467
　データメンバ ……467
　メンバ関数 ……468
　フレンド ……468
　関連する関数 ……469
TPReadObjects ……469
　メンバ関数 ……469
　フレンド ……470
TProgInit ……470
　メンバ関数 ……471
TProgram ……471
　データメンバ ……472
　メンバ関数 ……473
　パレット ……480
TPWObj ……483
　フレンド ……484
TPWrittenObjects ……484
　メンバ関数 ……484
　フレンド ……485
TRadioButtons ……485
　メンバ関数 ……486
　関連する関数 ……487
　パレット ……487
TRect ……487
　データメンバ ……488
　メンバ関数 ……488
　関連する関数 ……490
TResourceCollection ……490

データメンバ ……491
メンバ関数 ……491
関連する関数 ……493
TResourceFile ……493
データメンバ ……494
メンバ関数 ……494
TScreen ……499
データメンバ ……499
メンバ関数 ……500
TScrollBar ……502
データメンバ ……502
メンバ関数 ……504
関連する関数 ……507
パレット ……507
TScroller ……508
データメンバ ……508
メンバ関数 ……509
関連する関数 ……512
パレット ……512
TSItem ……513
データメンバ ……513
メンバ関数 ……513
TSortedCollection ……514
メンバ関数 ……514
関連する関数 ……516
TStaticText ……516
データメンバ ……517
メンバ関数 ……517
関連する関数 ……518
パレット ……518
TStatusDef ……519
データメンバ ……519
メンバ関数 ……520
TStatusItem ……520
データメンバ ……521

メンバ関数 ……522
TStatusLine ……523
データメンバ ……523
メンバ関数 ……524
関連する関数 ……527
パレット ……527
TStreamable ……527
メンバ関数 ……528
フレンド ……529
TStreamableClass ……529
メンバ関数 ……530
フレンド ……531
TStreamableTypes ……531
メンバ関数 ……531
TStringCollection ……532
メンバ関数 ……532
関連する関数 ……534
TStringList ……535
メンバ関数 ……536
関連する関数 ……537
TStrListMaker ……537
メンバ関数 ……537
関連する関数 ……539
TSystemError ……539
データメンバ ……539
メンバ関数 ……539
TTerminal ……540
データメンバ ……540
メンバ関数 ……541
フレンド ……543
パレット ……543
TTextDevice ……543
メンバ関数 ……544
パレット ……544
TView ……545

データメンバ……545
メンバ関数……550
フレンド……569
関連する関数……569
TVMemMgr……569
データメンバ……570
メンバ関数……570
TWindow……572
データメンバ……572
メンバ関数……574
関連する関数……578
パレット……578
TWindowInit……579
メンバ関数……579

第14章　エディタクラス……581
TEditor……582
TEditor のバッファ……582
ブロックの選択……584
オプション……584
キー割り当て……585
定数……585
データメンバ……587
メンバ関数……591
フレンド……605
関連する関数……605
TEditWindow……606
データメンバ……606
メンバ関数……606
関連する関数……608
TFileEditor……609
データメンバ……609
メンバ関数……609
関連する関数……612
TIndicator……613

データメンバ ……613
メンバ関数 ……613
関連する関数 ……614
TMemo ……615
メンバ関数 ……615
関連する関数 ……616

第 15 章　標準ダイアログボックスの実現 ……617
TChDirDialog ……617
コマンド定数 ……618
メンバ関数 ……618
関連する関数 ……621
TDirCollection ……621
メンバ関数 ……621
関連する関数 ……624
TDirEntry ……624
メンバ関数 ……625
TDirListBox ……625
メンバ関数 ……626
関連する関数 ……627
TFileCollection ……628
メンバ関数 ……628
関連する関数 ……632
TFileDialog ……632
オプション ……632
コマンド定数 ……633
データメンバ ……633
メンバ関数 ……634
関連する関数 ……636
TFileInfoPane ……637
データメンバ ……637
メンバ関数 ……637
関連する関数 ……638
TFileInputLine ……639
メンバ関数 ……639

関連する関数 ……640
TFileList ……641
コマンド定数 ……641
メンバ関数 ……642
関連する関数 ……644
TSortedListBox ……645
メンバ関数 ……645

第16章　グローバルリファレンス ……647

sample 関数 ……647
apXXXX 定数 ……647
Boolean 列挙型 ……648
BUILDER typedef ……648
bfXXXX 定数 ……649
ccAppFunc typedef ……649
ccIndex typedef ……650
ccNotFound 定数 ……650
ccTestFunc typedef ……650
cmXXXX 定数 ……651
cstrlen 関数 ……655
ctrlToArrow 関数 ……656
DEFAULT_SAFETY_POOL_SIZE ……656
dmXXXX 定数 ……657
EOS 定数 ……658
eventQSize 定数 ……658
evXXXX 定数 ……658
focusedEvents 定数 ……660
genRefs 関数 ……660
getAltChar 関数 ……660
getAltCode 関数 ……661
gfXXXX 定数 ……661
hcXXXX 定数 ……662
historyAdd 関数 ……663
historyCount 関数 ……663
historyStr 関数 ……663

inputBox 関数 ……………………………………………………………………663
inputBoxRect 関数 ………………………………………………………………664
kbXXXX 定数 ………………………………………………………………………664
lowMemory 関数 …………………………………………………………………667
maxCollectionSize 変数 …………………………………………………………667
maxFindStrLen 定数 ……………………………………………………………667
maxReplaceStrLen 定数…………………………………………………………668
maxViewWidth 定数 ……………………………………………………………668
mbXXXX 定数 ……………………………………………………………………668
message 関数………………………………………………………………………669
messageBox 関数 …………………………………………………………………669
messageBoxRect 関数……………………………………………………………670
mfXXXX 定数 ……………………………………………………………………670
newStr 関数 ………………………………………………………………………671
ofXXXX 定数 ……………………………………………………………………672
operator ＋（＋ 演算子） ………………………………………………………674
operator delete（delete 演算子） ……………………………………………674
operator new（new 演算子） …………………………………………………674
positionalEvents 定数 …………………………………………………………675
sbXXXX 定数 ……………………………………………………………………676
selectMode 列挙型 ………………………………………………………………677
sfXXXX 定数 ……………………………………………………………………677
specialChars 変数…………………………………………………………………680
StreamableInit 列挙型……………………………………………………………680
TScrollChars typedef ……………………………………………………………680
uchar typedef………………………………………………………………………681
ushort typedef ……………………………………………………………………681
wfXXXX 定数 ……………………………………………………………………681
wnNoNumber 定数 ………………………………………………………………682
write_args 構造体…………………………………………………………………683
wpXXXX 定数 ……………………………………………………………………683

索引 ……………………………………………………………………………685

はじめに

高い機能性を備え，フレキシブルで一貫性のある対話型ユーザインターフェースが要求されるキャラクタベースのアプリケーションを開発したいのであれば，Turbo Vision をお勧めします。

Turbo Vision は，DOS ベースのアプリケーションを対象とするアプリケーションフレームワークで，アプリケーション開発をまったく新しい見方で捉えます。Turbo Vision を使うことで，ウィンドウ，ダイアログボックス，メニュー，マウスをサポートする完全なユーザインターフェースを作成することができ，またそれぞれのアプリケーション用の単純な高速エディタを作成することも可能です。

このマニュアルでは，Turbo Vision で何ができるのか，どのようにすればよいのか，という点とともに，なぜそうなのかという点についても説明します。Turbo Vision の根底にある原理まで理解できれば，Turbo Vision が開発時間の節約をもたらす生産性の高いツールであり，時間をかけて理解する価値のあるものであるということがおわかりになることと思います。それまで思っていた以上に短い時間で，洗練された一貫性のある対話型のアプリケーションを作成できるようになるのです。

なぜ Turbo Vision なのか？

Turbo Vision とオブジェクト指向プログラミングを活用すれば，車輪を作り直す必要はありません。われわれの車輪を継承すればよいのです。

ボーランドでは，一貫性のあるユーザインターフェースを備えた数々のプログラムを作成してきましたが，今回これらの機能を再利用可能なツールセットとしてパッケージ化することにしたわけです。オブジェクト指向プログラミングがその手段となり，Turbo Vision はその結果です。

役立つかって？ もちろんです。事実 Turbo Pascal 6.0 の統合開発環境は，Turbo Vision を利用して作られたものであり，それも一から作り始めればかかるであろう時間の何分の一かの時間で作成されたのです。今度は，みなさんが自分のアプリケーションを開発するときに，この同じツールを利用できるわけです。

Turbo Vision とは？

Turbo Vision は，アプリケーションプログラムを開発する際に，基本的なプラットフォームを際限なく作り出していかなくてもよいようにします。

Turbo Vision は，ウィンドウ型アプリケーションの作成に必要となる，以下のような基本的な機能を提供する完全なオブジェクト指向クラスライブラリです。

- ■リサイズとオーバラップが可能な複数枚のウィンドウ
- ■プルダウンメニュー
- ■マウスサポート
- ■ダイアログボックス
- ■組み込みのカラーインストレーション
- ■ボタン，スクロールバー，入力ボックス，チェックボックス，ラジオボタン
- ■キー入力，マウスクリックの標準処理
- ■その他

Turbo Vision を利用すれば，あらゆるアプリケーションにおいて，みなさんの労力を極めて少ないものとしながら，一貫性のある最先端のルックアンドフィールを実現することができます。

知っておかなければならないこと

Turbo Vision を利用するためには，オブジェクト指向プログラミングおよび特に C++にかなり慣れておく必要があります。
Turbo Vision を利用して書かれたアプリケーションは，継承や多態性などの，さまざまなオブジェクト指向プログラミング上の技法を駆使することになります。

このマニュアルの内容

Turbo Vision は新規に開発されたツールであり，また Turbo Vision で使用されている技法は，おそらく多くのプログラマにとってなじみの薄いものであろうということを考慮に入れて，このマニュアルでは例題をたくさん取り上げています。また，リファレンスの章も設けています。

- ■第 1 部では，Turbo Vision の根底にある基本原理を紹介するとともに，Turbo Vision アプリケーションの開発プロセスを追っていくチュートリアルを示しています。
- ■第 2 部では，Turbo Vision クラス階層中のメンバなど，Turbo Vision の本質的な要素について，さらに詳しく説明しています。また，どうしたらアプリケーションをうまく記述できるようになるかについてのヒントも紹介しています。
- ■第 3 部は，Turbo Vision ヘッダファイルに含まれるすべてのクラスおよびその他の要素を参照するためのリファレンスです。

Turbo Vision のインストール

Turbo Vision には，INSTALL という自動インストールプログラムが添付されています。ファイル圧縮を行なっているので，このINSTALL プログラムが必要となります。Turbo Vision のファイルを単にハードディスクにコピーすれば済むというわけではないからです。INSTALL は Turbo Vision のファイルをコピーした後，自動的に解凍します。確認のため，インストール用ディスク中のREADME ファイルに，マスターディスクに含まれているファイルのリストを入れてあります。

このマニュアルでは，すでに DOS コマンドを熟知していることを前提としています。たとえば，マスターディスクのバックアップコピーを作成したい場合，DISKCOPY コマンドを使用する必要があります。マスターディスクを受け取ったら作業用のコピーを作成し，マスターディスクは安全な場所に保管しておいてください。

ボーランドのソフトウェアには，コピープロテクトがなされていません。弊社のライセンス契約（Borland's No-Nonsence License Statement）について詳しくご存じない方は，Turbo Vision のパッケージに同封されている同意書をお読みください。また，弊社宛にソフトウェア登録カードを必ず返送してください。このカードを返送していただけば，Turbo Vision の最新のアップグレード内容やバージョンについていち早くお知らせいたします。

INSTALL プログラムの使い方

インストールを行なう前に，README ファイルを読むことをお勧めします。

INSTALL は，まずみなさんがどのようなハードウェアを使用しているかを検出し，それに応じて Turbo Vision の各種設定値を決めます。また必要に応じてディレクトリを作成し，マスターディスクからハードディスクにファイルを転送します。INSTALL の処理内容はあらためて説明するまでもないでしょう。必要なことは以下のとおりです。

Turbo Vision をインストールするには次のようにしてください。

1. インストール用ディスク（ディスク1）をフロッピードライブにセットし，次のコマンドを入力して**リターン**キーを押してください（ディスクをセットしたドライブに従ってドライブ名を指定してください）。

```
A:INSTALL
```

2. インストール画面で**リターン**キーを押してください。
3. 以下，プロンプトメッセージにしたがってください。

README ファイルと HELPME!ファイル

重要！ インストールが終了したら，INSTALL プログラムは，Turbo Vision についての重要な最新情報を収めている README ファイルを読み込み，Turbo Vision についての新しい情報があることを紹介します。HELPME!.DOC ファイルがある場合，このファイルが技術サポート全般についての様々な質問に答えてくれます。

README ファイル

README ファイルにアクセスするには，次のようにしてください。

1. まだ Turbo Vision をインストールしていない場合は，ディスクをフロッピードライブ（仮にドライブ名を A とします）に挿入してください。すでに Turbo Vision のインストールが済んでいる場合は，3 に進むか，先に進んでください。
2. A: とタイプして**リターン**キーを押してください。
3. README とタイプして**リターン**キーを押してください。ファイルに入ったら，↑キーと↓キーでファイルをスクロールしてください。
4. 終了するときには，***ESC*** を押します。

Turbo Vision のインストールが済んだら，以下の手順にしたがって，README ファイルを Borland C++ （Turbo C++）の

IDE の Edit ウィンドウにオープンすることができます。

1. コマンドラインに BC (TC) とタイプして IDE を始動した後，**リターン**キーを押してください。
2. ***F10*** を押します。File | Open を選択してください。README とタイプインして**リターン**キーを押してください。
3. README ファイルを読み終えたら，IDE を終了して，Turbo Vision を使ってみてください。

HELPME!.DOC ファイル

Turbo Vision に, HELPME!.DOC というファイルを添付している場合があります。この場合そのファイルには，ユーザが一般に陥るトラブルに対する答えが入っています。何かトラブルが起きたら，このファイルを読んでください。README プログラムを使って HELPME!.DOC を読み出すことができます。コマンドラインに次のようにタイプしてください。

```
README HELPME!.DOC
```

このマニュアルで使う書体

`Monospace type`	この書体は，画面やプログラムで使用されるテキストそのものを表わします。また，入力そのものを表わす場合にも使用します。
ALL CAPS	定数名やファイル名は，大文字だけで表わします。
[ ]	テキストや DOS のコマンドラインにある大かっこ([ ])は，システムによって異なるオプション要素を表わすために使用されます。**このようなテキストが，そのまま入力されることはありません。**
Bold-Italics	Turbo Vision 標準の関数，クラス，および構造体の名前は，本文中ではボールドイタリック体で表記しています（プログラムの中ではボールドイタリック体にはなっていません)。
Italics	イタリック体は，本文中でユーザ定義の関数，クラス，および構造体を示すために使用しています。これらの単語は，そのままの名前で使うことも，新しい名前に変えて使うこともできます。また標準関数の引数もイタリック体で表わしています。
Boldface	Borland C++ 標準の関数名（**priutf** , **puts** など）を表わすために使用されます。
Gothic	Borland C++ の予約語（int , private）を表わすために使用されます。
Keycaps	この書体はキーボードのキーを示します。たとえば，「***ESC*** を押してメニューを終了してください」というようなときに使用します。

製品に関するお問い合わせ

ソフトウェアおよびマニュアルに関して，ご質問・ご意見などがありましたら，下記のサポートセンターまで，お電話，ファクシミリ，または文書にてお問い合わせください。なお，お電話での受け付け時間は，平日の 10：00〜12：00，および 13：00〜17：00 となっています（土曜，日曜，祝祭日は休業とさせていただきますのでご了承ください)。

〒150 東京都渋谷区恵比寿西 2-6-11 川上ビル
ボーランド株式会社
カスタマーサポートセンター
(TEL) 03-5489-6991
(FAX) 03-5489-4873

また，パソコン通信によるサポート窓口として，MSA ネット(同封の MSA ネット入会申し込み書をご覧ください)，日経 MIX(日経 BP 社の商用ネットワーク）の MSA ベンダ会議の 2 つの窓口を設けておりますので，営業時間内にご連絡いただけない場合などにご利用ください。

お問い合わせの際には下記の事項をお知らせください。

シリアル番号をお知らせいただかないとお答えできません。

■製品名とバージョン番号（Turbo Vision for C++ 1.0 など)
■ハードウェア環境：コンピュータ名とモデル番号（PC-9801 DA21 など)，実装メモリの種類と容量，周辺機器の接続，など
■ソフトウェア環境：MS-DOS および Windows のバージョン，Windows の動作モード，常駐プログラムやデバイスドライバ（RAM ディスク，EMS ドライバ，日本語 FEP）の設定状況，など
■各設定ファイル（AUTOEXEC.BAT，CONFIG.SYS，TPW.INI，WIN.INI）の内容

ご質問いただく前に，必ず以下の点についてご確認ください。

■操作上の問題が起きた場合は，Windows の他のタスクを終了

させてから再度実行してみてください。また，他の動作モード（スタンダードまたは 386 エンハンス）に変えて試してみてください。もし，特殊な拡張ボードを使用している場合は，それを外して試してみてください。可能であれば，他機種での実行結果もご報告ください。

■サポートセンターでは，ご報告いただいた問題を再現する必要がありますので，どのような状況で問題が発生するかをできるだけ詳細にご報告ください。自作のものや，特殊なデバイスドライバを組み込まれている場合は，サポートセンターでは対応できない場合がありますので，あらかじめご了承ください。

どうしても問題が解決されない場合は，その問題が発生する簡単なプログラムリストをお送りいただくこともできます。このとき，問題の発生と無関係な部分は削除して，問題を発生させるための簡単なプログラムとしてください。リストが 30～40 行以上になる場合や，読み込むべきデータファイルなどがある場合は，必ずプログラムやデータをフロッピーディスクに入れて，問題が再現できるようにしたうえでお送りください。上記のパソコン通信を利用できる場合は，そちらをご利用ください。プリンタ出力された大きなプログラムをお送りいただいても対処できませんのでご了承ください。お送りいただいたフロッピーディスクは原則としてお返しいたしません。内容を抹消する必要がある場合は，その旨を合わせてご連絡ください。

注意　言語自体のプログラミングについてのお問い合わせや，他機種や他の言語からの移植技法，Windows の内容や使い方，特殊な関数の作成方法などについてのご質問にはお答えできません。

第1部

はじめてのTurbo Vision

第1章
車輪の継承

みなさんが一番最近開発されたアプリケーションでは，肉の部分がどれだけで，骨の部分がどれだけだったでしょうか。

アプリケーションの中の肉の部分とは，そのアプリケーションが目的としている問題を解決する部分のことです。たとえば，計算，データベース処理などです。一方，骨の部分とは，肉の部分が強度と形状を維持するための基本的な構造です。たいていのアプリケーションでは，肉の部分を記述するのと同じぐらいの，あるいはそれ以上の時間を骨の部分の記述に費やしているのが現状です。この種のプログラムの基本構造は，一般に，**すべての**アプリケーションに応用できるにもかかわらず，習慣からか，新たなプロジェクトが始まるたびに，たいていのプログラマがフィールドエディタとかメニューマネージャとかイベントハンドラを，以前のものと大した差異はないのに，新規に開発しています。

みなさんは，同じ旧式の車輪を新たに作り直すことのないようにと，たびたび指摘されてきたわけですが，今回（Turbo Vision を利用することで）車輪の作り直しをやめにする機会が巡ってきたのです。これからは，車輪の**継承**を行なえばよいのです。

ウィンドウ型アプリケーションのフレームワーク

Turbo Vision は，イベント駆動によるウィンドウ型アプリケーションのフレームワークです。がっちりしていながらも柔軟な骨格で肉はほとんどついていません。C＋＋のオブジェクト指向プログラミングが備える拡張性を活用しながら，みなさんがこの骨格に肉をつけていくのです。Turbo Vision は，骨格となるアプリケーションクラス ***TApplication*** を用意していますので，みなさんは，特殊化したクラス（たとえば *TMyApplication* という名前とする）を導出することで，みなさんのアプリケーションをサポートすることができます。そうしておいて，*TMyAppl:cation* のメンバをふやしたりオーバーライドすることで，処理を行なうわけです。

アプリケーションの最上位で必要なのはこれだけです。アプリケーションプログラムの **main** 関数は，以下のような単純なものとなります。

```
int main()
{
   TMyApplication myApplication;      // インスタンスを作成し
   myApplication.run();               // それを実行する
   return 0;
}
```

オブジェクトを管理するためのこまごました処理，すなわちオブジェクトの初期化と廃棄については，舞台裏でコンストラクタとデストラクタが行なってくれます。

新しい視点から見たアプリケーション開発

C++の経験を積んだプログラマは，この節を読みとばしてもかまいません。

おそらくこれまでC/関数ライブラリを使用されたことがあるでしょうが，一見 Turbo Vision は，従来のライブラリによく似ているように見えます。究極的には，ライブラリというのは，メニューやウィンドウやマウスイベントハンドラなどをサポートするために使用されるものです。しかし，表面的には似ていても，その奥には根本的な違いがあります。この違いは理解しておく価値があります。それを理解することによって，高くて硬い概念的な壁をいくつか駆け上らなくても済むことになります。

まず第一には，今オブジェクトの世界にいるのだということを銘記しておかねばなりません。従来の構造化プログラミングにおいては，メニューマネージャなどのツールがまったく用をなさない場合には，そのツールのソースコードに手を加えて，要求どおりのものにすることになります。そのようなツールのソースコードの中に入り込んで，修正を加えるのは，大胆なやり方であり，そのコードがもともとどんなものであったのかを何らかの形で正確に把握しておかないかぎり，逆戻りするのが困難です。また，実際に稼働しているプログラムのソースコード（とくに他人の書いたソースコード）に手を加えることは，問題なく稼働していたサブシステムに新たなバグをいとも簡単にもたらすことにつながり，そのバグは，みなさんが対象としていた範囲をはるかに越えて波及していく可能性があります。

Turbo Vision を利用する場合，みなさんが既存のソースコードそのものに手を加える必要は決してありえません。Turbo Vision に“変更を加える”場合には，Turbo Vision を拡張するという形で行ないます。アプリケーションプロトタイプである ***TApplication*** は，TV.LIB 内部に隠されており，変更されることはありません。みなさんは，新しいクラス型を ***TApplication*** から導出することで ***TApplication*** を拡張し，導出したその新しいクラス用のメンバ関数を記述して，継承されたメンバ関数をオーバーライドすることで必要な変更を加えます。

また，**Turbo Vision は階層化されており**，ばらばらにして使用す

ることのできる単なるツールボックスとは異なります。したがって，その階層のどこかを使おうというのであれば，**全体で**使うことになります。Turbo Vision のすべての部品の背後には，アーキテクチャ上のビジョンは1つしかありません。それらの部品は，すべていろいろと複雑に連携しながら協調して動作します。マウスサポートだけを“ひっぱり出してきて”使おうとしてはいけません。そのような“つまみ食い”は，独自のマウスイベントハンドラを一から作り出す以上に大仕事となります。

➡ これら2つの注意すべき点，すなわち**オブジェクト指向の技法を存分に活用すること**と**Turbo Vision は，一まとまりとして利用すること**，が Turbo Vision における開発指針の基礎となります。つまり，Turbo Vision の“ルール”に従い，部品としてのクラスを意図された使い方に基づいて使用するということです。われわれが Turbo Vision を開発したのは，ユーザが膨大な作業を不必要に何度も繰り返して行なわなくてもよいように，実証済みの信頼できるアプリケーションフレームワークを提供するためです。Turbo Vision を実際に使ってみて，最大限に活用してみてください。

Turbo Visionのアプリケーションにおける各要素

Turbo Visionのアプリケーションがどのような動作をするかを見る前に，“ボックスの中に何があるのか”──アプリケーションを作成する際に，Turbo Visionのどのようなツールを利用できるのか──について説明しておきましょう。

部品の名前付け

Turbo Visionのアプリケーションは，**ビュー**（view）と**イベント**（event）と**ミュートオブジェクト**（mute object）とからなる協同社会です。

ビュー

ビューについては，第4章で詳しく説明します。

ビューとは，画面上に現われるすべてのプログラム要素のことで，すべてオブジェクトです。Turbo Visionを語る場合，視覚的に表現されるものはビューです。フィールド，フィールドキャプション，ウィンドウの境界線，スクロールバー，メニューバーは，すべてビューです。またビューを組み合わせることによって，ウィンドウやダイアログボックスなど，もっと複雑な要素を作り上げることもできます。組み合わせて作られた「集合的な」ビューは，**グループ**と呼ばれ，１個のビューであるかのように動作します。実際，グループは，サブビューという別のビューを所有することのできる特殊化されたビューです。またグループが別のグループを所有することもでき，この場合ビューとサブビューを精巧に組み合わせたチェインが形成されることになります。この章の後のほうで，アプリケーションそのものとなるグループにビューを挿入する方法についても説明しています。

ビューの形は，常に矩形です。この矩形には，１個分の文字を囲む矩形，あるいは１個分の文字の高さまたは幅を示す線分も含まれます。

イベント

イベントとは，何かが発生したということであり，アプリケーションはその出来事に対して処理を行ないます。イベントは，キーボードから発生したり，マウスから発生したり，Turbo Vision 内の他の部分から発生したりします。たとえば，キー入力やマウスボタンのクリックはイベントです。イベントは，それが発生した時点で，Turbo Vision のアプリケーションスケルトンによってキューにつながれ，イベントハンドラによって順番に処理されていきます。ユーザアプリケーションのフレームワークを用意する ***TApplication*** クラスにイベントハンドラが組み込まれています。イベント処理のメカニズムについては後で説明しますが，***TApplication*** で処理されないイベントは，そのイベントを処理するビューが見つかるか，“廃棄イベント（abandoned evnet）”エラーが発生するまで，そのプログラム内の別のビューに渡されていきます。

イベントについては，第5章で詳しく説明します。

たとえば，***F1*** キーの入力はヘルプシステムを起動しますが，どのビューもそのヘルプシステムに対する処理ルーチンを用意していない場合(コンテキストに応じたヘルプシステムなどの場合)，***F1*** キーの入力はメインプログラムのイベントハンドラによって処理されます。これに対して，通常の英数字キーや行内編集用のキーについては，その時点で**フォーカス**を合わせているビューが処理しなければなりません。

ミュートオブジェクト

ミュートオブジェクトとは，そのプログラムにおいてビューで**ない**オブジェクト全部のことです。“ミュート（無言）”なのは，それらのオブジェクト自体では画面に対して何ら語りかけないからです。これらのオブジェクトが行なうことは，計算であったり，周辺との通信であったり，あるいは一般的にはそのアプリケーション自体の処理であったりします。ミュートオブジェクトが画面に何らかの表示を行なう必要があるときには，ビューの力を借りて行ないます。**ディスプレイにアクセスできるのはビューだけである**というこの考え方は，Turbo Visionのアプリケーションを秩序立

てる上で重要なことです。

☞ ミュートオブジェクトが**printf**文もしくは<<文を使用してディスプレイに表示しようとした場合には，それを防ぐ手だてはありません。“勝手に”ディスプレイに出力した場合には，そのテキストによってTurbo Visionが出力したテキストは乱されてしまうことがあり，またこの“背教徒”テキストは，Turbo Visionが出力したテキストによって（たとえば，ウィンドウを移動させたり，ウィンドウのサイズを変えたりしたときに），消されてしまう場合があります。

共通の“ルックアンドフィール”（外見と感触）

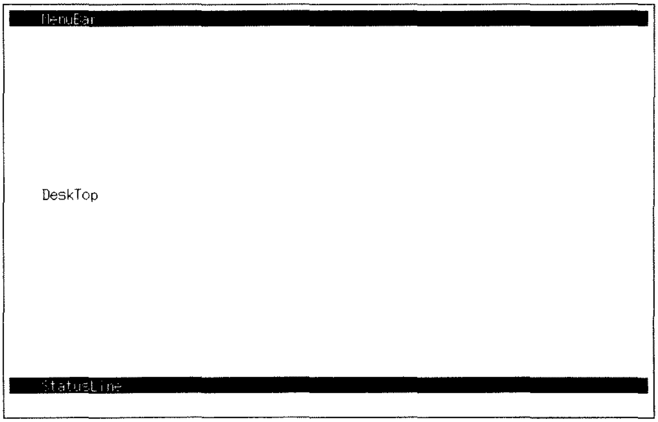
Turbo Visionは，画面の設計に対する標準的で合理的なアプローチとして開発されたものです。したがって，ユーザアプリケーションに親しみやすいルックアンドフィールを持たせることができます。このルックアンドフィールは，ボーランドのコンパイラで提供されているものと同じルックアンドフィールであり，何年にもわたる経験と使い勝手についてのテストとを経てきたものです。使い方のよくわかった共通の画面をアプリケーションに持たせることができるという特長は，アプリケーションユーザにとっても，アプリケーション開発者にとっても好都合なことです。アプリケーションの処理内容がどんなに難解なものであっても，その使用方法は常にわかりやすいものとなり，学習曲線を上昇させるのに役立つことになります。Turbo Visionは，このような標準的なインターフェースを用意しながら，ユーザ独自の設計はまったく自由に行なうことができるようにしています。

ここにでてくる各アイテムについては，第4章『ビュー』で説明します。

図1.1にTurbo Visionのアプリケーションで使われる共通のオブジェクトを示します。デスクトップとは，色つきの背景のことであり，その上にアプリケーションが表示されます。Turbo Visionの場合，ほとんどそうですが，デスクトップもオブジェクトです。ディスプレイの最上段にあるメニューバーも最下段にあるステータスラインも同様です。メニューバー上の単語は，メニューを表わします。その単語のところにマウスポインタをもってきてクリックするか，

またはホットキーを入力することで，メニューがプルダウンされます。

図1.1
画面上の Turbo Vision オブジェクト



ステータスライン上のテキストは，ユーザが自由に決めることができますが，通常は，ユーザアプリケーションの現在の状況を表示したり，有効なホットキーを示したり，その時点でユーザからのコマンドを受け付けることができることを示したりします。

プルダウンされたメニューでは，マウスの動きやカーソルキーの入力に応じて，メニューリスト中の対応する項目のところにハイライトバーがきます。**リターン**キーを押すか，マウスの左ボタンをクリックすると，その時点でハイライトしていた項目が選択されます。メニュー項目の選択により，アプリケーションのしかるべきルーチンにコマンドが伝えられます。

一般にユーザアプリケーションは，1個ないしは複数個のウィンドウまたはダイアログボックスを通してユーザとの交信を行ないます。ウィンドウやダイアログボックスは，マウスやキーボードからのコマンド入力に応じて，デスクトップ上で現われたり消えたりします。Turbo Vision では，情報を入力したり表示するためのさまざまな仕掛けをウィンドウに持たせています。ウィンドウ内部は，スクロールできるようになっており，ウィンドウは文書ファイルのような大きなデータを表示するための窓口の役割を果たすこともできます。これは，見方を変えれば，データに対してウィンドウを動かしていくということになります。ウィンドウ内のデータは，ウィンドウの最下段と右端にあるスクロールバーを動かすことによってス

クロールさせることができます。スクロールバーは，バックグラウンドビューに対するウィンドウの相対的な位置を示すものです。

ダイアログボックスには，ボタンを持たせることがよくあります。ボタンは，単語をハイライトさせたもので，それをクリックすることによって（あるいは，***TAB*** キーを用いてボタン間の移動を行ない，**スペース**キーを押すことで）選択することができます。クリックすると，（実際のプッシュボタンと同じように）その単語は“奥の方に”押されたような表示となり，アプリケーションに対してコマンドを送るように設定したことになります。

Turbo Vision 流 "Hello, World!"

Turbo Vision では従来とは異なる方法で "Hello, World ! " と表示します。

新しい言語やユーザインターフェース・ツールキットの使い方を紹介する場合，そのツールを使って記述された "Hello, World ! " プログラムを示してみせるというのが従来のやり方でした。そのようなプログラムは，画面上に "Hello, World ! " という文字列を表示し，DOS に制御をもどすだけのコードで充分でした。

古典的な "Hello, World ! " プログラムは，対話型には**なっていません**（"語り手" とはなっていても，"聞き手" とはなっていません)。これに対して Turbo Vision は，対話型プログラムを作り出すためのツールです。

"Hello, World ! " プログラムは，マスターディスクに HELLO.CPP というファイル名で入っています。

Turbo Vision のアプリケーションは，一番簡単なものでも，{と}で通常の **printf** を挟んだものより，かなり複雑なものとなります。古典的な "Hello, World ! " プログラムと比較した場合，Turbo Vision の HELLO.CPP は，以下に示すように，かなりいろいろなことを行なう必要があります。

- ■デスクトップをハーフトーンパターンにする。
- ■最上段にメニューバーを，最下段にステータスラインを表示する。
- ■キー入力およびマウス用のイベントハンドラを初期化する。
- ■メニューオブジェクトとして "隠れている部分" を作成し，それをメニューバーにリンクする。
- ■ダイアログボックスを作り，"隠れている部分" も作成しておく。
- ■ダイアログボックスをメニューにリンクする。
- ■ユーザが，マウスかキーボードから何かを入力するまで待ち続ける。

このリストには，画面にテキストを表示する処理は入っていません。何らかのテキストが用意されるのですが，それはすべてバックグラウンドにあり，コマンドが発生したときに初めて呼び出されるようになっています。このことは，みなさんが Turbo Vision を理解していく上で，念頭においておかなければならない点です。Turbo

Vision でのプログラミングの基本は，自分なりのビューを設計し，コマンドが渡されたときに何をさせるかを決める，ということです。Turbo Visionは，フレームワークとして，みなさんのビューにコマンドが正しく伝わるように手配します。みなさんは，キー入力やマウスクリックやメニューコマンドがみなさんのビューに渡ってきたときに何を行なわせるかをプログラムすればよいのです。

みなさんのプログラムの肉の部分というのは，ユーザが入力したコマンドに応じて，何らかの処理を行なうコード（プログラム）のことであり，この“肉に当たる”コードは，みなさんが作成したビューオブジェクト内に置かれます。

HELLO.CPP を動かしてみよう

HELLO.CPP の内容を詳しく解析していく前に，まずプログラムをロードし，コンパイルしてみて，どのような動きをするかを見てみましょう。

HELLO を実行させると，画面を初期化し**図 1.2** にあるようなデスクトップを作り上げます。ウィンドウは，まだ何もオープンされておらず，画面最上段のメニューバーに **Hello** というコマンドだけが示されています。**Hello** の“**H**”は，“ello”とは別の色になっています。またステータスバーには，“GRPH-X Exit”というメッセージが表示されています。

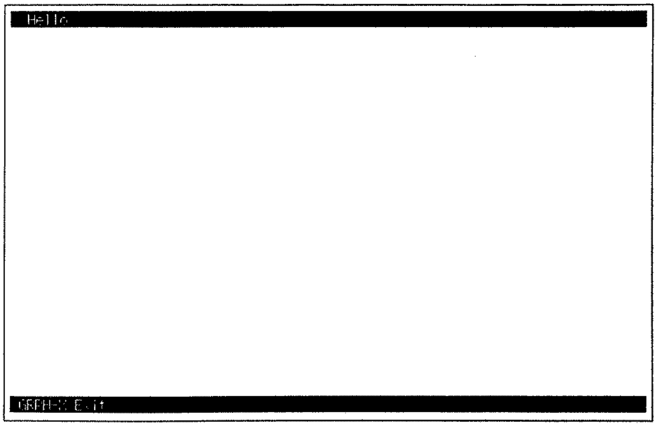


図 1.2
HELLO.CPP の初期画面

ここで，**すべての**ユーザ環境において注意すべきプログラミング上の原則を２つ取り上げておきましょう。

1. **ユーザが次に何をすればよいのかわからないような状態にしてはならない。**
2. **ユーザがいつでも先に進んだり，後戻りしたりできるようにしておかねばならない。**

HELLO を動かしたユーザは，まだ何も行なっていないこの時点では，２つのことを選択できることが容易にわかります。すなわち，メニュー項目 **Hello** を選択するか，***GRPH-X*** をキー入力してこのプログラムから抜け出すという選択です。

メニューをプルダウンする

以上のことを頭に入れて，メニューバーの **Hello** を選択してみましょう。選択方法は３つあります。

- ■マウスポインタを **Hello** の上にもっていき，左ボタンをクリックする。
- ■ ***F10*** キーを押して，メニューバーのところにカーソルをもっていく。そうすると，**Hello** が反転するので，そのときに**リターン**キーを押して **Hello** を選択する。
- ■ ***GRPH-H*** を押す。H というのは，メニューコマンド **Hello** のうちハイライトした文字のこと。

いずれの場合でも，項目 **Hello** の下にプルダウンメニューが現われます。これは，ボーランドの統合環境（IDE）と同じ動作であり，IDE を使用しているプログラマなら見慣れた画面だと思います。Turbo Vision では，何もかもをボーランドの IDE（統合開発環境）に倣っているということがおいおいわかってくることと思います。

図 1.3 が出てくるメニューです。このメニューでは，項目は２個だけで，１本の線で２つの**枠**に分けられています。HELLO の場合は，簡単なプログラムであるため，それぞれの枠には１個のメニュー項目しかありませんが，実際には，画面の大きさが許すかぎり，１つの枠にいくらでも項目を並べることができます。

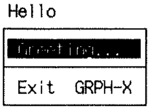


図 1.3
HELLO.CPP のメニュー

メニュー項目は，キーボードからでもマウスからでも選択できます。カーソル移動キーで，メニュー内のハイライトバーを上下ドラッグさせることができます。キーボードからある項目を選択したい場合には，その項目がハイライトにされている状態で**リターン**キーを入力します。マウスで選択する場合には，少し違った趣向になっています。マウスポインタがハイライトバー上にあるときにマウスの左ボタンを押し，そのまま押したままにしておくことで，ハイライトバーを“意のままに動かす”ことができます。マウスボタンを押している間は，メニュー項目のリスト内でハイライトバーを上下（ドラッグ）させることができます。メニュー項目を選択するには，選択したい項目のところにハイライトバーをもってきてマウスボタンを離します。

ダイアログボックス

メニュー項目の後にある省略符号（...）は，その項目がダイアログボックスまたはさらに下位のサブメニューを起動することを示すものです。

Greeting の項目を選択すると，今度は**図 1.4** にあるようなダイアログボックスと呼ばれるウィンドウが現われます。ダイアログボックスは，画面の中央に現われますが，画面内でドラッグすることもできます。ダイアログボックスを動かすには，マウスポインタをダイアログボックスの上枠にもってきて，マウスの左ボタンを押したままマウスを動かします。ボタンを離すと，ダイアログボックスの位置が決まります。

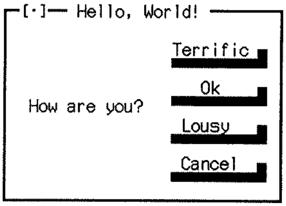


図 1.4
Hello,World! のダイアログボックス

このダイアログボックスには，“Hello World!”というタイトルがついており，左上隅にはクローズアイコンがあります。クローズア

イコンは，それをマウスでクリックすると，ダイアログボックスをクローズさせ，画面から消し去ります。ダイアログボックス内には，簡単なテキスト“How are you?”が表示されています。これは，固定文字列と呼ばれるものであり，表示はされるけれども，対話には使われないテキストです。すなわち，固定文字列は，何かのラベルとしては使われるけれども，それをクリックしても何も引き起こさないテキストのことです。

ボタン

ダイアログボックス内で右側に並んでいる4個の矩形が，この“Hello, World!”ダイアログボックスで一番重要な部分です。これは，ボタンと呼ばれるもので，コントロールの一種です。コントロールと呼ばれるのは，電気製品のつまみ（control）に似た働きをするからです。どのボタンにもラベルがついていて，そのボタンが押されたときに何を行なうのかが示されています。

ボタンを押すには，そのボタン上でマウスをクリックするか，またはそのボタンをデフォルト状態（これについては，以下で説明します）にして**リターン**を入力します。マウスでボタンをどれか押してみて（ボタンの上にマウスポインタをもってきて，マウスボタンを押す)，どうなるか調べてみてください。ボタンの内容が1文字分右にずれて，影がなくなるのがわかると思います。四角いボタンが画面の“奥の方向に”押されているかのように見えるのがおわかりでしょう。マウスボタンを離すと，そのボタンで定義されていた動作が起こります。

モノクロのシステムでは，デフォルトボタンを“〉〈”の文字で表わします。

Cancelボタンは，他のボタンとは別の色で表示されています。色を違えてあるのは，このダイアログボックスで現在Cancelボタンがデフォルトコントロールになっていることを示します。Cancelがデフォルトになっているときに**リターン**を入力すると，Cancelボタンが押されたことになります。

ダイアログボックスのデフォルトコントロールは，***TAB***キーを押すことで変えることができます。“Hello, World!”ダイアログボックスで，***TAB***を入力してデフォルトコントロールをひと通り変えてみてください。デフォルトを示す色が，***TAB***キーを押すごとに，あ

るボタンから次のボタンに移っていくのがわかります。こうすれば，マウスを使わなくてもボタンを押すことができます。***TAB*** キーを使って，選択したいボタンをデフォルトにし，**リターン**キーか**スペース**キーを押すことで，“ボタンを押す”のと同じことができるわけです。

ダイアログボックスがアクティブになっているときには，メニューバーをマウスでクリックしても何も起こらないことに注意してください。ダイアログボックスに制御が渡されているからです。

抜け出し方

HELLO ダイアログボックスのボタンをどれか押すと，ダイアログボックスは消え去り，ただのデスクトップにもどります。再度 Hello メニューをプルダウンさせ，ダイアログボックスを出現させることもできます。何回でも。プログラムから抜け出るには，Hello メニューで Exit 項目を選択するか，単に Exit のショートカット ***GRPH-X*** を入力します。ショートカットの表示は，Helloメニュー内部にも画面最下段のステータスラインにもあります。

☞ **ユーザがいつでもプログラムから簡単に抜け出せるようにする**というのが良い方法です。出口を見つけられずにどうしてよいかわからないユーザは，すぐにシステムをリブートするものです。そうすると，アプリケーションがファイルをクローズしたり，シャットダウン時の終了処理を行なったりできなくなります。

HELLO.CPP の内部構造

以上が HELLO の動作です。さて，この動作はどのようにして実現されているのでしょうか。HELLO プログラムの多くは，というよりほとんどが，Turbo Vision で用意されている標準クラスを継承したものです。継承されたものばかりなので，最初のうちは，実行時にそれが**どのように**動作するのかが少し謎めいて見えます。統合デバッガで実行を追ってみても，Turbo Vision はコンパイル済みのライブラリなので全体図がなかなか摑めません。ただ，何が行なわれているかが理解できるようになると，正確な**仕組み**を知る必要はなくなります。

Turbo Vision のアプリケーションを理解するには，まず，**Turbo Vision のアプリケーションが，たくさんのオブジェクトの協同作業によって成り立っている共同体である**ということを念頭に置くことです。まずは主要なオブジェクトをいくつか見つけ，それらのオブジェクト間でどのような動作が行なわれているのかを理解します。その後で，こまごまとしたオブジェクトが主要なオブジェクトをどのようにサポートしているのかを見ていけばよいのです。

実際の具体的な定義を読み取る前に，＊.H にあるクラス宣言を読み取り，理解するようにしてください。まず，オブジェクトに何が含まれ，そのオブジェクトがシステム内の他のオブジェクトとどういう関係にあるのかを理解することが重要です。

アプリケーションクラス

すべてのアプリケーションの基礎となるのが ***TApplication*** クラスです。といっても，***TApplication*** 型のオブジェクトを直接インスタンス化することはありません。***TGroup*** の孫である ***TApplication*** は，それぞれのアプリケーションにおいて，メニューやウィンドウなどのサブビューからなるビューの動的なチェインを所有したり制御したりすることのできるグループとなっています。***TApplication*** は，それだけでアプリケーションとなるわけではあ

りません。***TApplication*** は基本クラスとして使用されるのです。そしてこの基本クラスから，プログラムの肉の部分となるクラスを導出します。それぞれのプログラムには，このアプリケーションクラスのインスタンスは 1 つしか必要ありません。HELLO の場合，その導出クラスを *THelloApp* とし，以下のように宣言します。

```
class THelloApp : public TApplication
{

public:

    THelloApp();           // コンストラクタ

    virtual void handleEvent( TEvent& event );
    static TMenuBar *initMenuBar(TRect r);
    static TStatusLine *initStatusLine(TRect r);

private:

     void greetingBox();
};
```

もっと複雑なプログラムでは，クラスの宣言は別個の .H ファイルで行ない，実際の定義は .CPP ファイルで行ないます。

もちろん，*THelloApp* には，そのコンストラクタと上記 4 種類のメンバ関数以外にも，数多くのメンバが含まれます。導出クラスは，直接の基本クラスからすべてを継承するからです（その基本クラスは，さらにその基本クラスから継承する，というように継承を重ねてきています）。*THelloApp* を作成する際，新しいクラスが，親である ***TApplication*** とどう**違う**かを定義します。再定義しないものについては，すべて ***TApplication*** からそのまま継承されます。

コンストラクタについての説明は少し後回しにし，*THelloApp* に定義されている 4 種類のメンバ関数について考えてみましょう。これらの関数はアプリケーション全体の「概要」を決めるものです。

■アプリケーションの動作は，そのアプリケーションがどのイベントをどのように処理するかで決まります。この最も重要な仕様を実現するのが ***handleEvent*** であり，これを定義する必要があります。仮想メンバ関数 ***handleEvent*** は，***TApplication*** から継承され，どのようなアプリケーションでも発生するような一般的なイベントを処理するようになっています。が，アプリケーションに固有なイベントを処理させるために，必ずオーバーライドする必要があります。

- 静的メンバ関数 ***InitMenuBar*** は，***TMenuBar*** オブジェクトへのポインタを返します。この関数は，アプリケーションのメニューバーからプルダウンされるメニューを作成します。***TApplication*** では，メニューバーは定義していますが，メニューは定義していません。メニューを使えるようにするには(メニューを使わないものとすれば，アプリケーションは何とも貧弱なものになってしまう！)，***initMenuBar*** などのメンバ関数を定義することでメニューを用意しなくてはなりません。後で説明しますが，***THelloApp*** コンストラクタが ***initMenuBar*** を呼び出して作成した ***TMenuBar*** オブジェクトをアプリケーショングループに挿入します。
- 同様に，静的メンバ関数 ***initStatusLine*** は，画面の最下段に表示されるステータスラインのテキストを用意し，***TStatusLine*** オブジェクトへのポインタを返します。通常このテキストでは，アプリケーションのそのときの状態やホットキーを表示したり，ユーザに次の操作を指示したりします。***initMenuBar*** の場合と同様，*THelloApp* コンストラクタが ***initStatusLine*** を呼び出して ***TStatusLine*** オブジェクトをアプリケーショングループに挿入します。
- *greetingBox* プライベートメンバ関数は，メニュー項目 Greeting に対するダイアログボックスを作成します。*greetingBox* は，マウスまたはキーボードでメニュー項目 Greeting を選択することで発生するイベントに基づき，***handleEvent*** メンバ関数内から呼び出されます。高度なアプリケーションだと，最初のメニューで定義されるメニュー項目ごとにメンバ関数を分けたりします。

要約すれば，*THelloApp* の各メンバ関数は，メインプログラムの各オブジェクトが果たすべき機能を決めます。すなわち，アプリケーションを初期化するためのコントラスタ，関係するイベントを認識するための“エンジン”(イベントハンドラ)，およびそれぞれのイベントを処理するためのメンバ関数を定義します。一般に，この3つが，***TApplication*** の導入クラスに付加すべき機能です。

ダイアログボックスオブジェクト

以上のオブジェクトの他にHELLOで使われる主なオブジェクトは，ダイアログボックスのオブジェクトだけです。ダイアログボックスは，特別に何かを行なわなければならないというわけではないので，HELLOでは，***TDialog***クラスのインスタンスを利用しています。***TDialog***から特別なクラスを導出する必要はありませんが，そうしたほうが便利な場合もあります。

TDialog自体には，対話を行なう機能は何もなく，単なるフレームです(といっても賢いフレームです)。したがって，ユーザとの対話に使われるフィールドやコントロールは，すべて，プログラマが作成することになります。

*THelloApp::greetingBox*は，やはりTurbo Visionのビューの一種であるボタン（画面に**何か**を表示するオブジェクトは，**すべて**Turbo　Visionのビューです！）を4個挿入するだけで，あとは***TDialog***に任せます。これが，ダイアログボックスの一般的な使い方です。通常，ダイアログボックス内に持たせたいコントロールを挿入するだけです。これ以外にダイアログボックスに必要なものは，すべて(そのイベントハンドラも含み)，***TDialog***の中で作成されます。

実行の流れとデバッグ

Turbo Visionのアプリケーションは，イベント駆動型のプログラムになりますので，従来のプログラムとは少し構造が異なります。とくにイベント駆動型のプログラムでは，ユーザからの入力（あるいはその他のイベント）を読み出して評価する制御構造と，その入力データを基にして処理を行なう関数とは分離されます。

従来のプログラムの場合，入力データをいくらか読み込み，そのデータをどのルーチンで処理するかを決め，そのルーチンを呼び出し，さらにこの一連の手順を繰り返す，といったブロックをいろいろと連ねるというのが一般的です。さらに，入力データの処理が終

わったら，次の入力データを処理するためには，どこに制御を渡さなければならないかを，いちいち決めておく必要があります。

これに対してイベント駆動型プログラムでは，中心にイベントディスパッチ機構があり，それ以外の大半の部分は，データを入力したり，そのデータをどこで処理するかを決める，といったことに煩わされずに済むようになっています。各ルーチンは，処理すべき入力データがディスパッチャから渡されてくるのを待てばよいのです。このことは，デバッグの方法にも大きな変化をもたらします。おそらくみなさんのデバッグの進め方も変わることになります。まずイベントハンドラにブレークをかけて，メッセージのディスパッチがうまくいっているかどうかをチェックし，次にイベント処理ルーチンにブレークをかけて，それがちゃんと動作しているかどうかをチェックする，といったものになります。

Turbo Vision アプリケーションのデバッグについては，第10章『ヒントとコツ』で，細かなテクニックを紹介します。

HELLO のメインプログラム

Turbo Vision ではすべてのアプリケーションが，だいたい以下の HELLO と同じような **main** プログラムをトップレベルにおくことになります。

```
int main()
{
   THelloApp helloWorld;
   helloWorld.run();
   return 0;
}
```

それでは，**main** の本体について，もう少し詳しく見てみましょう。

アプリケーションのインスタンス化

最初の文は，*helloWorld* という *THelloApp* のインスタンスを宣言しています。これにより，コンストラクタ *THelloApp::THelloApp* が呼び出され，オブジェクト *helloWorld* の作成と初期化を行ないます。ここでは細かい説明は省きますが，THelloApp コンストラク

タは，Turbo Vision のクラス階層を ***TObject***（Turbo Vision の「ルート」クラス）から ***TView***，***TGroup***，***TProgram***，***TApplication***，そして最終的には *THelloApp* まで基本クラスのコンストラクタを呼び出していきます。

舞台裏で行なわれるこうした一連の処理により，様々なデフォルト状態とメカニズムがセットアップされ，ほとんどのアプリケーションに対して標準インターフェースが用意されることになります（上級レベルのプログラマが，標準装備のものとは別のインターフェースを作成したいというのであれば，必要なクラスのコンストラクタが第 13 章で詳細に説明してあるので，それを研究してオーバーライドしてください）。この結果，メインプログラムのオブジェクト ***helloWorld*** は，消去済みの画面全体からなるデスクトップビューを得て活動を開始します。***TProgram*** コンストラクタは，***initMenuBar*** および ***initStatusLine*** を呼び出し，特定の HELLO メニューとステータスラインを作成して挿入します。基本的な手順は，以下のとおりです。

```
// TProgram および TProgInit のコンストラクタの一部

menuBar = initMenuBar();    // menuBar のポインタを獲得し
if ( menuBar != 0 )
   insert( menuBar );  // app グループのサブビューリストに挿入

statusLine = initStatusLine();  // statusLine のポインタを獲得し
if ( statusLine != 0 )
   insert( statusLine );    // app グループのサブビューリストに挿入
```

この段階で，これらの動作の全体像がわかるはずです。*THelloApp::initMenuBar* と *THelloApp::initStatusLine* のソースコード（HELLO.CPP 内にある）を見れば，基本的な方法がわかります。第 2 章で，メニューバーとそれにリンクされるメニューおよびコマンドの作成法，ならびにステータスラインとそのテキストおよびコマンドの作成方法について詳しく説明します。

デバッガを使って HELLO.EXE をステップ実行し，そのときの表示を確かめてみるとよいでしょう。コンストラクタが呼び出されてから，デスクトップ，メニューバー，ステータスラインが画面上に配置されていき，*helloWorld.run* から利用できるようになるのがわかります。

run メンバ関数

Turbo Vision アプリケーションのほとんどの謎は，メインプログラムの ***run*** メンバ関数に集中しています。謎は，まず *THelloApp* の中で ***run*** の定義を探そうとする時点から始まります。そこには ***run*** メンバ関数はありません。というのは，***run*** は *THelloApp* の基本クラス ***TApplication*** から，その基本クラス ***TProgram*** を介してそのまま継承されているからです。***TProgram::run*** は ***execute*** を呼び出し，おそらくこれがアプリケーションの中で大半の時間を消費することになるでしょう。以下の簡略化したコードで示すように，***execute*** は do..while ループを二重にネストさせています。

```
ushort TGroup::execute()
{
    do {
       endState = 0;
       do {
          TEvent event;
          getEvent( event );
          handleEvent( event );
          } while ( endState == 0 );
    } while( !valid(endState) );
    return endState;
}
```

イベントの処理の仕方については第 5 章で詳しく説明します。

詳細は省いてありますが，Turbo Vision アプリケーションは 2 つのタスクの間をループすることがわかります。すなわち，***getEvent*** を使ってイベントを取得すること（ここでイベントとは，「何か処理すべきこと」です），および ***handleEvent*** を使ってそのイベントを処理することの，2 つのタスクです。最終的には，イベントが何らかのプログラム終了コマンドに還元されたときにループが終了します。

アプリケーションデストラクタ

HELLO.CPP には，とくにデストラクタは定義されていません。

プログラムが終了すると，オブジェクト *helloApp* とそれにリンクされているすべてのビュー（メニューバー，ステータスライン，デスクトップ）は，コンストラクタが呼び出されたときとは逆の順序で，その基本クラスのデストラクタを自動的に呼び出すことによって廃棄されていきます。最後に，Turbo Vision のエラーハンドラとドライバがシャットダウンされます。一般に，Turbo Vision のオブジェクトを廃棄するための特殊なコードを記述する必要はありません。

まとめ

この章では，Turbo Vision がどんなものであるかを簡単に味わってもらいましたが，楽しんでいただけたでしょうか。イベント駆動型のフレームワークの中で，オブジェクトが相互作用する様子を見て，Turbo Vision がどんなツールを用意しているのかがある程度理解できたことと思います。

これまでの説明を読んでみて，ひとつ HELLO.CPP を書き換えて，何か他のことをやらせてみようという気になってきたのではないでしょうか。遠慮なくそうしてみてください。Turbo Vision が素晴らしいのは，ほとんど手を加えることなく自由にプログラムを書き換えることができるということです。

次の章では，Turbo Vision で用意される骨組みを基にして，自分でプログラムを作成する方法を紹介します。

第2章
Turbo Visionアプリケーションの作成

前章で，Turbo Visionのアプリケーションがどんなものであるかについて，内側からも外側からも見てきたので，自分でもアプリケーションを組んでみたいと，うずうずしているのではないでしょうか。この章では，まさしくそのアプリケーションの作成を行ないます。まず，簡単なフレームワークから初めて，ステップを追いながら少しずつプログラムを書き足していきます。

おそらく質問したいこともたくさんあると思います。ビューはどのような動作をするのか，グループはサブビューをどのようにして制御するのか，ビューを使って何ができるのか，ビューを自分のアプリケーションに組み込むには，どうすればよいのか。Turbo Visionが従来どおりのランタイムライブラリなら，ソースコードをほじくりかえしてその答を得ようとするかもしれません。

しかし，Turbo Visionは既成のアプリケーションです。Turbo Visionについてのそのような疑問に答えるには，実際にビューを使ってみるのが最善の方法です。すぐにわかることですが，ビューの初期化は，ほんの少しのプログラムで済むのです。

最初の Turbo Vision アプリケーション

Turbo Vision のアプリケーションを開発する場合，いつでも，***TApplication*** から導出されたクラスをインスタンス化することから始まります。

以下の例では，***TApplication*** から *TMyApp* というクラスを導出します。そしてまずその中で，***TApplication*** のメンバ関数をオーバーライドしたり新しくメンバを追加したりします。次に，*myApp* という *TMyApp* 型のオブジェクトを宣言します。*myApp* はグループという特殊なビューであり，階層をさかのぼっていくと ***TApplication***, ***TProgram***, ***TGroup*** および ***TView*** の各クラスを祖先としています。これらのクラスから，*myApp* はいろいろな性質と機能を継承します。継承したそれらの性質と機能は，プログラムで明示的に指定しなくても利用することができます。Turbo Vision を利用してプログラミングを行なう場合には，各クラスにどのような「デフォルト」機能が付加されているのかを理解する必要があります。この章を読んでいただければ，*myApp* のグループとしての性質がとくに重要であることがおわかりになると思います。最終的にはアプリケーションのグループは，一連のビューを所有することになり（これらのビュー自身がサブビューを所有する別のグループであったりする)，それらのビューがユーザからの「ランダムな」イベントを処理することになります。

☞

プログラムには，TApplication のオブジェクトは1つしか存在しません。

以後，この章では，多くの場合，*myApp* を例題として使用していきます。*myApp* がみなさんのアプリケーションであり，***TApplication*** の下位クラスのインスタンスです。みなさんが Turbo Vision のアプリケーションを作成する場合には，そのアプリケーションの機能に相応した名前をつければよいのですが，“the instance of the object you derived from TApplication（***TApplication*** から導出したクラスのインスタンス)”というのでは長ったらしいので，ここでは *myApp* という名前にします。

マスターディスクには，段階をおって，例題プログラムのいくつかのバージョンが納められています。ファイル名は，各プログラムリストの左欄に示してあります。

まずは，以下に示すプログラム例でプログラム作成を開始してみましょう。プログラムを変更するたびに，そのリスト全体を示すようなことはしません。追加したり，変更したりした部分だけを掲載します。ここで紹介する変更を逐一追いながら，実際に変更を行なえば，少しずつ機能が上がっていくのがわかると思います。自分なりにプログラムを変更してみるのも，非常に有益だと思います。またマスターディスクにある各種ヘッダファイル（*.H）を調べてみるのもよいでしょう。これらのヘッダファイルには，クラス宣言，メンバ関数のプロトタイプ，データメンバ定義，および各種定数およびマクロが入っています。第12章のヘッダファイルのクロスリファレンスも参照してください。

TVGUID01の（あるいは，Turbo Vision のアプリケーションの場合いつでも）メインブロックは，以下のようなものになります。

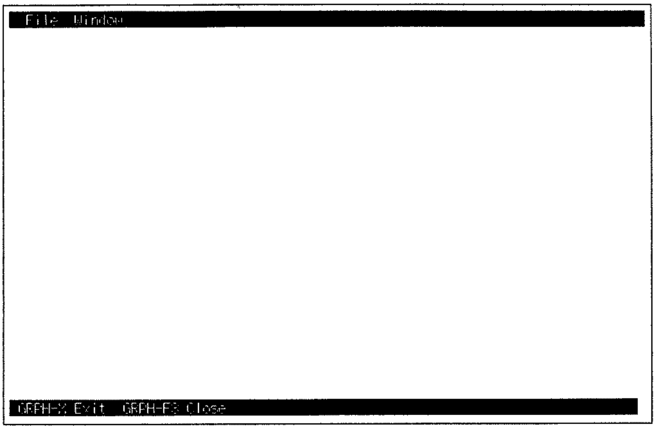
これは，マスターディスクに，例題プログラムの1つとして，TVGUID01.CPP という名前で格納されています。

```
// TVGUID01.CPP

#define Uses_TApplication
#include <tv.h>

class TMyApp : public TApplication
{

public:
   TMyApp();

};

TMyApp::TMyApp() :
   TProgInit( &TMyApp::initStatusLine,
          &TMyApp::initMenuBar,
          &TMyApp::initDeskTop
          )
{
}

int main()
{
   TMyApp myApp;
   myApp.run();
   return 0;
}
```

ここでは，まだ *TMyApp* に新しい機能を追加していません。通常，メンバを新たに追加したり，継承したメンバをオーバーライド

することなく導出クラスを宣言しても，ほとんど意味がありません。***TApplication*** のインスタンスとして，オブジェクト *myApp* を宣言するだけでよいのです。後々 Turbo Vision のアプリケーションの開発を進めていくにつれ，*myApp* に機能を追加していくことになりますので，*TMyApp* は将来の追加作業の出発点であるとみなすことができます。さしあたっては，“プレーンバニラのような”単なる ***TApplication*** としての役割しか果たしません。***TApplication*** オブジェクトがデフォルトで行なうことは，**図 2.1** にあるような画面を作り出すことです。

図 2.1
デフォルトの
TApplication 画面



このプログラムは，ちゃんと動きますが，単に ***GRPH-X*** の入力に対してプログラムを終了するだけの働きしか持っていません。他に何かやらせたければ，デフォルトの機能以外にも，ステータスラインかメニューバーにテキストとコマンドを追加する必要があります。次の節でこれを行ないます。

デスクトップ，メニューバー，ステータスライン

使用するクラス
TView
TGroup
TMenuBar
TMenuBox
TStatusLine
TProgInit
TProgram
TApplication
TDeskTop

TVGUID01.CPP では，*TMyApp* コンストラクタの本体は定義されていません。*TMyApp* コンストラクタは基本クラスである ***TProgInit*** のコンストラクタを呼び出します。***TProgInit*** は ***TProgram*** の仮想基本クラスです。***TProgram*** コンストラクタは ***TGroup*** コンストラクタを呼び出します。

```
TProgram::TProgram() : TGroup(/* ビュー境界に関する引数 */)

// TGroup コンストラクタは全画面ビューを作成する
{
   ...
   if( createStatusLine != 0 &&
       (statusLine = createStatusLine( getExtent() ) != 0 )
       insert( statusLine );
   ...
// createStatusLine は実際には TProgram (オーバーライドされて
// いれば最も下位の導出クラス) で定義されている initStatusLine
// を呼び出す. 第13章の TProgInit コンストラクタを参照.

// statusLine は，新たに作成されたステータスラインオブジェクト
// を指す静的データメンバ. insert() はアプリケーショングループ
// にステータスラインを挿入して表示する.

// getExtent() は，それを呼び出したオブジェクトの矩形境界を返す
    ...
// createMenuBar, createDeskTop 用に同様なコードを加える
    ...
}
```

これら 3 つの関数ポインタ ***createXXX*** には，それぞれ，対応する ***initXXX*** 関数のアドレスがセットされます。これに関しては, TVGUID02 と第 13 章の *TMyApp* および ***TProgInit*** のコンストラクタの項で詳細を説明しています。

TVGUID01.CPP の何も飾り気のないデスクトップ, メニューバーおよびステータスラインは，***TProgram*** コンストラクタがデフォルトのメンバ関数 ***initDeskTop***, ***initMenuBar*** および ***initStatusLine*** を呼び出すことによって作成され，*myApp* グループに挿入されます。***TApplication*** は直接の基本クラスである ***TProgram*** からこれらの関数をそのまま継承します。もちろん,

TMyApp も ***TApplication*** からこれらの関数を（今のところ）そのまま継承します。***TProgram*** で定義されているように，この 3 つの ***initXXX*** 関数は図 **2.1** に示される最小限のビューを作成します。この素気ないインターフェースで満足だというのであれば，継承した ***initXXX*** にそれぞれの仕事を行なってもらうだけで充分でしょう。実際，デフォルトのデスクトップは，ほとんどの目的を果たしてくれるので，通常 ***initDeskTop*** をオーバーライドする必要はありません。しかし，メニューバーとステータスラインに関しては，アプリケーションごとに肉付けを行なう必要があるため，常に ***initMenuBar*** と ***initStatusLine*** をオーバーライドする必要があります。その方法については，以下で説明していきます。

TV.H の使用法については，第 12 章で詳しく説明します。

#define Uses_TApplication とそれに続けて #include <tv.h> を記述することによって，***TProgram***，***TProgInit*** および ***TApplication*** の各クラスを宣言したインクルードファイル APP.H を取り込むことができます。例題プログラムが大きくなり，導入する標準クラスをふやしたい場合には，クラスに対応する #define Uses_TClassname 文を追加していけばよいのです。TV.H は，*TClassname* の .H ファイルだけでなく，*TClassname* のすべての基本クラスに対応する .H ファイルを重複することなく適切にインクルードするためのものです。

TProgram は，***initDeskTop***，***initMenuBar***，および ***initStatusLine*** を呼び出して，***TProgram*** の静的データメンバである ***deskTop***，***menuBar***，および ***statusLine*** に，それぞれ対応するオブジェクトへのポインタをセットします。このことは，アプリケーションが，いつでもデスクトップ，メニューバー，ステータスラインのオブジェクトを 1 つずつしか持つことができないということを意味します。それぞれについて見てみることにしましょう。

デスクトップ

initDeskTop は，以下に示すように ***TDeskTop*** オブジェクトを作成して，そのオブジェクトへのポインタを返します（細かいことについては，気に留める必要はありません。***TRect*** の意味は，説明が先に進むにつれて明らかになります)。

```
TDeskTop *TProgram::initDeskTop(TRect r)
{
    r.a.y++;            // 左上の y 座標を調整
    r.b.y--;            // 右下の y 座標を調整
                        // メニューバーとステータスラインの分を
                        // 除いた全画面を示す
    return new TDeskTop( r ); // デフォルトのデスクトップを作成
}
```

デスクトップへのポインタ ***deskTop*** は，デスクトップがアプリケーションごとに1個しか存在しないので，***TProgram*** の静的なデータメンバとなっています。デスクトップが ***TProgram*** コンストラクタによってアプリケーションのグループに挿入されることについては，先に説明しました。グループ *myApp* は，これでデスクトップオブジェクトを所有することになります。しかしデスクトップもまたグループです。デスクトップは *myApp* によって所有されますが，デスクトップ自身がビューを所有することもできます。実際デスクトップはグループとしてデスクトップ上に表示されるすべてのビューを管理しますので，アプリケーションで重要な役割を果たします。たとえばユーザがメニュー選択項目をクリックした場合，*myApp* は適当なサブビューをインスタンス化してデスクトップグループに挿入しますが，デスクトップはそのサブビューおよびそのサブビューでさらにサブビューが生成された場合にはそのサブビューも管理します。ここでいう管理とは，イベント処理およびイベントに応じたビューの表示・消去・サイズの変更などのホスト処理のことです。

ステータスライン

TProgram コンストラクタは，***TProgInit::createStatusLine*** を呼び出して，***TStatusLine*** オブジェクトを作成し，それへのポインタ ***statusLine*** を返します。実際のステータスラインは，***TProgram::initStatusLine*** をオーバーライドして，それぞれのアプリケーションで必要とされるものを作成するようにします。その方法については，もう少し後で説明します。みなさんは，自分の *TMyApp::initStatusLine* のアドレスを ***TProgInit*** コンストラクタに引き渡すだけで，後は Turbo Vision に任せればよいのです。

deskTop の場合と同様，***statusLine*** は ***TProgram*** の静的なデータメンバです。つまり，***TProgram*** クラスのすべてのオブジェクトに関して，インスタンスは1個だけです。ステータスラインのおもな機能は，ホットキーの定義と表示です。(ホットキーとは，シフトキー，コントロールキー，***GRPH*** キーおよび ***^KF*** キーを含めたキー入力のことであり，メニューやステータスラインの項目と同様の機能を果たします。)

ステータスラインは，画面の左端から表示されます。画面の最下行の中でステータスラインの項目として使用されない部分は，すべて他のビューのために解放されています。****statusLine*** は，ホットキーにコマンドを割り当てます。ステータスラインの項目自体は，マウスの左ボタンでクリックすることもできます。***initStatusLine*** の概要を捉えることができるように，以下に ***TProgram*** で定義されているデフォルトのものを示しておきます。これは，**図 2.1** にあるような，「***GRPH***-X」だけの最小限のステータスラインを作成します。その他 ***F10, GRPH-F3, F5,*** および ***CTRL-F5*** をそれぞれ IDE の標準コマンド ***cmMenu, cmClose, cmZoom,*** および ***cmResize*** に割り当てていますが，"GRPH-X Exit" 以外は，テキストを何も表示しないようにしています。この時点では，***cmQuit*** コマンドだけがアクティブ（有効）となっています。

```
TStatusLine *TProgram::initStatusLine(TRect r) {
    r.a.y = r.b.y - 1;
    return new TStatusLine( r,
        *new TStatusDef( 0, 0xFFFF ) +
```

```
        *new TStatusItem( "~GRPH-X~ Exit", kbAltX, cmQuit ) +
        *new TStatusItem( 0, kbF10, cmMenu ) +
        *new TStatusItem( 0, kbAltF3, cmClose ) +
        *new TStatusItem( 0, kbF5, cmZoom ) +
        *new TStatusItem( 0, kbCtrlF5, cmResize )
        );
}
```

#define Uses_TClassname の中には余分なものもありますが，残しておいても問題はありません。むしろ用心に越したことはありません。

次に，ステータスラインにテキストを追加しながら，***initStatusLine*** の構文について紹介します。TVGUID02.CPP では，***TApplication::initStatusLine*** をオーバーライドして，先とは異なるステータスラインを作成します。TVGUID02 の始めに #define Uses_TClassname の行が追加されていることに注意してください。新しいクラスを使用するごとに1行追加されています。たとえば Uses_TKeys は，***kbF5*** や ***kbAltF3*** などのキーボードニーモニックを使用できるようにするためのものです。

次に，***initStatusLine*** 宣言を ***TMyApp*** に追加し，さらに以下の定義も追加します。

これは，TVGUID02 CPP の一部です。

TMyApp の宣言部に，static TStatusLine * initStatusLine(TRect r); を追加することを忘れないでください。

```
static TStatusLine *TMyApp::initStatusLine(TRect r)
{
    r.a.y = r.b.y - 1;      // 最下行を最上行の1行下に設定
    return new TStatusLine( r,
        *new TStatusDef( 0, 0xFFFF ) +
        // ヘルプコンテキストの範囲を設定
        *new TStatusItem( "~GRPH-X~ Exit", kbAltX, cmQuit ) +
            // 項目を定義
        *new TStatusItem( "~GRPH-F3~ Close", kbAltF3, cmClose )
            // 項目をもう1つ定義
        );
}
```

Turbo Vision のコマンドは定数であり，識別子は "cm" で始まります。

初期化は，***TStatusDef*** コンストラクタと ***TStatusItem*** コンストラクタを続けて呼び出すことにより行ないます(詳細は第13章で説明)。オーバーロード演算子 **+** は，以下に示すように，***TStatusDef*** オブジェクトと ***TStatusItem*** オブジェクトからなるリンクリストを作成し，そのリストを ***TStatusLine*** のデータメンバ ***items*** に渡すために使用されます。

```
TStatusDef& operator + (TStatusDef& s1, TStatusItem& s2);
```

以下が，***TStatusItem*** コンストラクタのプロトタイプです。

```
TStatusItem::TStatusItem(const char *aText, ushort key, ushort cmd,
                         TStatusItem *aNext = 0);
```

見てのとおり，ステータス項目は，テキスト文字列（不要ならヌルにする），キーコード，コマンドコード，および次のステータス項目へのポインタ（ヌルの場合はそれ以上項目がないという意味）を指定します。

TVGUID02 の ***TStatusDef*** オブジェクトは，ヘルプコンテキスト 0～0xFFFF の範囲で表示されるステータスラインを定義しています。また Turbo Vision の標準コマンド ***cmQuit*** を ***GRPH-X*** に，***cmClose*** を ***GRPH-F3*** のキー入力に対応させています。

上の例では，最後のステータス項目は，テキスト引数として "~GRPH-F3~ Close" をとっています。文字列中，チルダ (~) で囲まれた部分は，画面上ではハイライトします。ユーザは，この文字列中のどこかにマウスをもってきて左ボタンをクリックすることで，そのコマンドを実行することができます。

TVGUID02 を実際動かしてみると，ステータス項目 ***GRPH-F3*** がハイライトしておらず，それをクリックしても何も起こらないことがわかります。これは，***cmClose*** コマンドがデフォルトで無効状態にされているからです。無効なコマンドを発生する項目も無効状態になるわけです。一度ウィンドウを開いてやると，***cmClose*** もそのステータス項目も，ちゃんと働くようになります。

これで *TMyApp* コンストラクタの定義は終わりましたので，以下のように，***TProgram*** で定義されているデフォルトの ***initStatusLine*** ではなく，独自の ***initStatusLine*** を呼び出します。

```
TMyApp::TMyApp() :
   TProgInit( &TMyApp::initStatusLine, &TMyApp::initMenuBar,
              &TMyApp::initDeskTop )
{
}

/*
3つの initXXXX 関数のアドレスを TProgInit のコンストラクタに渡
す．このコンストラクタは，createStatusLine などの3つの creatXXXX
関数ポインタを初期化し，それらを使ってアプリケーションのステータス
ライン，メニューバー，およびデスクトップを初期化する．
*/
```

なぜ ***TProgInit*** というクラスがあるのかについては，第 13 章で技術的な説明を行なっています。簡単に説明するなら，***TProgram*** は ***TGroup*** と ***TProgInit*** の 2 つのクラスを基本クラスとしています。***TProgInit*** は，***createStatusLine***，***createDeskTop***，および ***createMenuBar*** の 3 つの関数へのポインタを持っているパブリックの仮想基本クラスです。仮想基本クラスのコンストラクタには，独自の特別なルールがあります。仮想基本クラスのコンストラクタは，ほとんどの導出クラスから，すべての導出クラスのコンストラクタを呼び出す前に呼び出されます。したがって関数へのポインタ ***createStatusLine*** には *&TMyApp::initStatusLine* がセットされます。*TMyApp* は，***initMenuBar*** や ***initDeskTop*** を定義していないため，*&TMyApp::initMenuBar* と *&TMyApp::initDeskTop* は，継承してきた関数を示すことになります。

ここでは，定義済みのコマンド（***cmQuit*** と ***cmClose***）しか使用していませんので，***statusLine*** を初期化してしまえば，ステータスラインの作成作業は終わりです。***statusLine*** は，これだけで，ユーザ入力を処理できるようになります（***statusLine*** は静的なメンバですので，***TProgram*** の導出クラスでないクラスから ***statusLine*** を参照する場合には，修飾子付き形式 ***TProgram::statusLine*** を使用して参照する必要があります)。

新しいコマンドの作成

ステータスラインの各項目に対応づけられたコマンド ***cmQuit***，***cmClose*** は，Turbo Vision の標準コマンドでした。したがって，コマンドを定義する必要はありませんでした。自作コマンドを使いたい場合は，自作コマンドを定数として宣言します。たとえば，新しいウィンドウをオープンするためのユーザ独自のコマンドを定義するには，

Turbo Vision で定義済みのコマンドには，定数が予約されています。これについては，第 5 章の 184 ページを参照してください。

```
const cmNewWin = 199;
```

とします。

そして，ホットキーあるいはステータスラインの項目に，そのコマンドを対応させます。

```
return statusLine = new TStatusLine ( r,
*new TStatusDef(0, 0xFFFF ) +
     *new TStatusItem( "~GRPH-X~ Exit", kbAltX, cmQuit ) +
     *new TStatusItem( "~F4~ New", kbF4, cmNewWin ) +
     *new TStatusItem( "~GRPH-F3~ Close", kbAltF3, cmClose)
     );
```

cmNewWin にリンクする機能をどのようにして実現するかについては，後の章のイベント処理のところで説明します。

ステータスラインの初期化の書き方がわかれば，メニューの初期化も理解しやすいでしょう。ただしメニューは，ステータスラインと似ていますが，もう少し複雑になります。

メニューバー

TProgram コンストラクタから呼び出されるデフォルトの ***initMenuBar*** は，以下のようにして ***TMenuBar*** を作成し，静的データメンバ ***menuBar*** の設定を行ないます。

```
TMenuBar *TProgram::initMenuBar( TRect r )
{
    r.b.y = r.a.y + 1;        // 最下行の y 座標を最上行の
                              // 1行下に設定
    return new TMenuBar( r, 0 );
}
```

TVGUID01 で見たように，このデフォルトの ***initMenuBar*** は，メニュー項目が何もリンクされていない空のメニューバーを作成します。第1引数は，ビューのサイズ（矩形領域）を指定します。***new TMenuBar*** の第2引数にある 0 ポインタは，何もメニュー項目がないこと，すなわち何もテキストが表示されないことを意味します。独自のメニュー階層を作成するには，***initMenuBar*** をオーバーライドする必要があります。***menuBar*** は，ステータスラインの場合と同様にオーバーロード演算子 **+** を使用して，***TMenuItem*** コンストラクタと ***TSubMenu*** コンストラクタをネストさせて呼び出すことによって初期化します。またステータスラインの場合と同様，

```
static TMenuBar *initMenuBar( TRect r);
```

を *TMyApp* クラスで宣言しておく必要があります。

関数 ***newLine*** を使用すれば，メニュー項目の間に仕切り線を引くことができます。メニューを初期化してしまえば，メニュー作成作業は終わりです。あとは，メニューバーがユーザ入力の処理を行ないます。

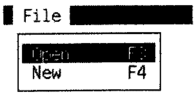
選択項目が1個しかないメニューを，1つだけサポートする単純なメニューバーの場合，初期化は次のようになります。

```
const cmFileOpen = 200; // 新しいコマンドを定義

TMenuBar *TMyApp::initMenuBar(TRect r)
{

    r.b.y = r.a.y + 1;     // 最下行を最上行の1行下に設定
    return new TMenuBar( r,
    *new TSubMenu("~F~ile", kbAltF ) +
    *new TMenuItem( "~O~pen", cmFileOpen, kbF3, hcNoContext, "F3" )
    );
}
```

このプログラムでは，“File”というメニューが作成され，選択項目は“Open”です。チルダ（~）で，“File”および“Open”のショートカット文字を，それぞれ**F**および**O**にします。また，**F3**キーを“Open”のホットキーにします。

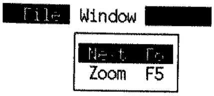
Turbo Visionでは，すべてのビューにヘルプコンテキスト番号というものが付けられます。この番号は，アプリケーション全体を通して，コンテキスト（状況）に応じたヘルプ表示ができるようにするものです。ビューは，デフォルトで ***hcNoContext*** というコンテキストをもっています。これは，現在のコンテキストを変更しないという特殊なコンテキストです。メニューバーにヘルプコンテキストを付加する必要がでてきたら，***initMenuBar*** ルーチンの中で ***hcNoContext*** を別の値に変えてやればよいでしょう。

“File”メニューに2番目の項目を追加するには，以下のように，****new TMenuItem*** 呼び出しをもう1つネストさせます。




```
return new TMenuBar( r,
  *new TSubMenu("~F~ile", kbAltF ) +
  *new TMenuItem( "~O~pen", cmFileOpen, kbF3, hcNoContext, "F3" )+
  *new TMenuItem( "~N~ew",  cmNewWin,   kbF4, hcNoContext, "F4" )
  );
```

また，2個目のメニューを追加するには，以下のように，***new TSubMenu*** 呼び出しをもう1つネストさせます。

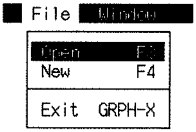



```
return new TMenuBar( r,
    *new TSubMenu( "~F~ile", kbAltF )+
    *new TMenuItem( "~O~pen", cmFileOpen, kbF3, hcNoContext, "F3" )+
    *new TMenuItem( "~N~ew",  cmNewWin,   kbF4, hcNoContext, "F4" )+
    *new TSubMenu( "~W~indow", kbAltW )+
    *new TMenuItem( "~N~ext", cmNext,     kbF6, hcNoContext, "F6" )+
    *new TMenuItem( "~Z~oom", cmZoom,     kbF5, hcNoContext, "F5" )
    );
```

ここでは，さらに Turbo Vision の標準コマンドである ***cmNext*** と ***cmZoom*** の2個のコマンドを，メニュー項目およびホットキーに割り当てています。

メニュー選択項目の間に仕切り線を入れたい場合には，以下のように，***new TMenuItem*** 呼び出しの間に ***newLine*** 呼び出しを入れます。

(TVGUID03.CPP)：




```
return new TMenuBar( r,
   *new TSubMenu("~F~ile", kbAltF)+
   *new TMenuItem("~O~pen", cmFileOpen, kbF3, hcNoContext, "F3")+
   *new TMenuItem("~N~ew", cmNewWin, kbF4, hcNoContext, "F4")+
   newLine()+
   *new TMenuItem("E~x~it", cmQuit, kbAltX, hcNoContext, "GRPH-X")
   );
```

マスターディスク内の TVGUID03.CPP では，ステータスラインにステータスキーの追加も行なっています。**F10** キーを ***cmMenu*** コマンドに割り当てています。***cmMenu*** は Turbo Vision の標準コマンドで，マウスを使用しないユーザがメニューバーを扱えるようにします。その場合，**F10** キーを押すと，メニューバーがアクティブになり，カーソルキーを使って，メニューおよびメニュー項目を選択できるようになります。***handleEvent*** を変更する必要はありません。

また，**F10** キーに対応するステータス項目のテキストは空の文字列になっていて，画面上では何も表示されません。**F10** でメニューを有効にできることをユーザに知らせてもよいのですが，メニューを有効にするために項目をクリックするというのも意味がありません。それだったら，メニューバーを直接クリックしたほうがましです。

Turbo Vision フレームワークについての補足

ここまでのプログラムで，たくさんのコマンドが使えるようになりました。ただし，そのほとんどは無効状態になっています。また，*cmNewWin* および *cmFileOpen* コマンドは，まだ何も実行しません。

何かがっかりしたという人があるかもしれませんが，そう思うのは早計です。たくさんのことをやってきたわけですから。これまでみてきたことは，イベント駆動型プログラミングがもたらす大きなメリットの中の１つにしかすぎません。すなわち，**入力を行なう**機能と**入力に応じた処理を行なう**という機能とを分離できるということでした。

従来のプログラミング手法に依るならば，ウィンドウをオープンするといったプログラムを追加する場合，開発したばかりのプログラムをまた見直さねばならないことになります。しかし，それは必要ないのです。コマンドを生成してくれる確実なエンジンが，すでに掌中にあるのですから。必要なのは，生成されたコマンドに対応して処理を行なうルーチンをいくつか書くことだけです。次の節でそれを行ないます。

Turbo Vision のアプリケーションフレームワークを利用することにより，従来のモジュールプログラミングより，さらに一歩前進することができます。単にプログラムを再利用可能な機能ブロックに細分できるというだけではないのです。これらのブロックは，もっと小さくすることができ，さらに独立性と互換性を高めることができるのです。

ウィンドウをオープンするコマンド（*cmNewWin*）を発生させるには，いくつかの方法があることを見てきました。ステータスライン項目を使う方法，メニュー項目を使う方法，ホットキーを使う方法です。このコマンドが発生したら，アプリケーションにウィンドウをオープンするよう伝えなければなりませんが，ここで少し，それがいかに簡単であるかについて触れておきます。ここで一番大切なことは，コマンドが**どのようにして**発生したのかについては，アプリケーション側では関知しないということです。ウィンドウも関知しません。そのような仕事は，完全に独立して行なわれるのです。

後でコマンドの対応付けを変更したくなった場合，たとえばメニュー項目を移動するとか，ホットキーの割り当てを変更するといったことを行ないたい場合でも，それによって他のところにどう影響するのだろうかなどと考える必要はありません。これが，イベント駆動型プログラミングで得られるメリットなのです。ユーザインターフェースとプログラム自体の機能とを分けて考えることができるわけです。また，プログラムのそれぞれの部分は，まさしく独立に機能するようになるのです。

ウィンドウの操作

使用するクラス
TRect
TView
TWindow
TGroup
TScroller
TScrollBar

Turbo Vision のウィンドウはオブジェクトであり，利用する側でプログラムを何も書かなくても，たいていのユーザ入力を処理できるようになっています。ウィンドウは，外部で発生するイベントに応じてウィンドウ自体を，オープン，リサイズ，ドラッグ，ムーブ，およびクローズする方法を知っています。といってもプログラマがウィンドウ上に直接何かを書くわけではありません。ウィンドウは，ウィンドウ以外のオブジェクトを所有し，管理するグループですが，それらのオブジェクトを画面上に表示することはしません。ウィンドウは，ビューの管理を行なうのであり，アプリケーションそれぞれの機能は，ウィンドウが所有し，管理するビューを使って具現されます。皆さんが作り出したビューは，いろいろなところに，いろいろな方法で表示することができます。

それでは，ウィンドウ内に何かを表示させるには，標準ウィンドウツールをどのように組み合わせて利用すればよいのでしょうか。まずは，標準ウィンドウを作成し，それに必要な機能を付加していくことにしましょう。プログラム例をいくつか見ていくうちに，Turbo Vision で提供されている骨組みを利用すれば，簡単に肉付けしていくことができることがおわかりになることと思います。

次のプログラムは，ウィンドウを初期化し，デスクトップに貼りつけるものです。みなさんの *TMyApp* クラスの宣言に新しいメンバ関数を追加するのを忘れないようにしてください。ここでも，その基本クラス ***TWindow*** から継承したメンバに何もメンバを追加せずに，新しいクラス (*TDemoWindow*) を定義しています。以前にもそうしたように，後で簡単に拡張できるように，簡素なプラットフォームを作っておくわけです。それに新しいメンバ関数をどんどん追加していけばよいのです。

これは，TVGUID04.CPP です。

```
static short winNumber = 0;          // ウィンドウ番号を初期化

void TMyApp::handleEvent(TEvent& event)
{
  TApplication::handleEvent(event); // 基本クラスと同様に機能!
  if ( event.what == evCommand )
  {
```

```
        switch( event.message.command )
        {
           case cmMyNewWin: // ただし，追加コマンドに応答する
             myNewWindow(); // cmMyNewWin コマンドのアクションを定義
             break;
           default:
             return;
        }
        clearEvent( event );       // 処理したらイベントをクリア
    }
}

class TDemoWindow : public TWindow // 新しいウィンドウクラスを定義
{

public:

  TDemoWindow(const TRect& r, const char *aTitle, short aNumber);
// コンストラクタを宣言

};

TDemoWindow::TDemoWindow( const TRect& r, const char *aTitle,
                          short aNumber):
              TWindow( r, aTitle, aNumber),
              TWindowInit( &TDemoWindow::initFrame )
{
}

void TMyApp::myNewWindow()
{
   TRect r( 0, 0, 26, 7 );           // 初期サイズと位置を設定
   r.move( random(53), random(16) ); // 画面内をランダムに移動
   TDemoWindow *window = new TDemoWindow ( r, "Demo Window",
                                           ++winNumber);
   deskTop->insert(window); // ウィンドウをデスクトップに入れて表示
}
```

ハイレゾリューションモードでこの例題を実行する場合は，この行をr.move(random(53), random(22))；と変更してください(以降の例題も同様)。

プログラムからこのウィンドウを使えるようにするには，以前と同様，コマンド *cmNewWin* を，メニューオプションまたはステータスラインのホットキーに結合します。ユーザ入力により *cmNewWin* コマンドが呼び起こされると，Turbo Vision は，コマンドを *TMyApp::handleEvent* に送ります。すると *TMyApp::handleEvent* は，*TMyApp::myNewWindow* を呼び出します。

ウィンドウの構成

TRect クラスについては，第4章『ビュー』のところで詳しく説明します。

TWindow コンストラクタは，***TWindow*** オブジェクト自体を初期化するために引数を3つとります。大きさと位置，タイトル，ウィンドウ番号です。これによって，ウィンドウ自身が初期化を行います。

ウィンドウの大きさと位置を指定する1番目の引数は，***TRect*** です。これは，Turbo Vision の矩形領域クラスです。***TRect*** は，そのコンストラクタがウィンドウの左上隅と右下隅の座標（4つの整数座標または2つの ***TPoint*** オブジェクトにより表わされる）を指定することで，大きさと位置を決めるという簡単なクラスです。***TRect*** オブジェクトを操作したり，比較したりするために，いくつかのメンバ関数とオーバーロード演算子が用意されています。詳細については，第12章『**ヘッダファイルクロスリファレンス**』を参照してください。

TVGUID04 では，まず矩形 r をデスクトップの原点に作り，その後，デスクトップ上でランダムな距離だけ移動させます。通常のプログラムでウィンドウをランダムに移動させることはありませんが，ここでは，ウィンドウをいろいろな位置にたくさんオープンするための簡単な手法を示そうというわけです。

TWindow コンストラクタの2番目の引数は，ウィンドウのタイトルとして表示される文字列です。

最後の引数には，ウィンドウの ***number*** データメンバを指定します。***number*** が1から9までであれば，ウィンドウのフレーム上にその番号が表示され，ユーザが **GRPH-1** から **GRPH-9** を入力すれば，対応する番号のウィンドウを選択できるようになっています。

ウィンドウに番号を割り当てる必要がない場合には，定義済みの定数 ***wnNoNumber*** を指定します。

TDemoWindow コンストラクタは，***TWindow*** と ***TWindowInit*** の両コンストラクタを呼び出します。***TWindowInit*** は，***TProgInit*** と ***initStatusLine*** の説明にあったように，引数に ***&initFrame*** をとります。この時点では，まだ ***TWindow::initFrame*** をオーバーライドしていないので，標準フレームが表

示されることになります。***initFrame*** をオーバーライドして凝った表示にすることもできますが，ほとんどのアプリケーションでは，この標準フレームで充分用が足ります。

挿入メンバ関数

ウィンドウをデスクトップに挿入すれば，自動的にウィンドウが表示されます。***insert*** メンバ関数は，ビューの制御を別のビューに与えるときに使用します。たとえば，

```
deskTop->insert(window);
```

という命令を実行すると，*window* をデスクトップに挿入することになります。ビューは，いくつでもデスクトップのような**グループ**クラスに挿入することができます。ビューを挿入したグループのことを**オーナー**ビューといい，挿入されたビューのことを**サブビュー**といいます。サブビュー自身がグループであったり，サブビュー自身をサブビューとして持つこともできます。たとえば，ウィンドウをデスクトップに挿入した場合，このウィンドウはサブビューですが，ウィンドウ自体は，フレームやスクロールバーなどのサブビューを持つことができます。

こうしたビュー同士の関係については，第4章で詳細を説明します。

このようにして，ビュークラス間のリンクを行なうことによって，**ビューツリー**ができあがります。ビューツリーという名前は，ちょうど木の幹から枝が分かれていくように，アプリケーションを中心のビューとして，ビューやサブビューがたくさんリンクしながら枝別れしていくことからきています。

ウィンドウのクローズ

ウィンドウのクローズアイコンをクリックすることによって，***cmClose*** コマンドが発生します。このコマンドは，***GRPH-F3*** キー入力およびステータスライン項目に結合させたものと同じものです。ウィンドウをオープンすれば(***F4*** キーの入力，もしくは File | Open メニューの選択による)，デフォルトとして，自動的に，***cmClose*** コマンドならびに ***cmClose*** を発生するビューを実行可能状態にします（***cmZoom*** や ***cmNext*** などのウィンドウ関係のコマンドについ

ても同じ)。

ウィンドウをクローズするために，新たに何かプログラムを記述する必要はありません。ユーザがウィンドウのクローズアイコンをクリックするか，***GRPH-F3*** キーを押すか，あるいはステータス項目上をクリックすれば，後は Turbo Vision が処理を行ないます。ウィンドウが ***cmClose*** コマンドを受けると，デフォルトで，そのウィンドウのデストラクタを呼び出します。デストラクタは，ウィンドウのオブジェクトとタイトル文字列を消去します。またそのウィンドウが所有するすべてのサブビューのデストラクタの呼び出しも行ないます。ウィンドウのコンストラクタの中で，自分で余計にメモリ割り当てを行なった場合には，ウィンドウのデストラクタを適切にオーバーライドさせて自分でそのメモリを解放しなければなりません。

ウィンドウの動作

プログラムを実行させて，いろいろとさわってみてください。ここまでのものだけでも，いろいろな機能が備わっています。デスクトップ上の複数のウィンドウに対して，オープン，クローズ，選択，移動，リサイズ，ズームを行なうことができます。150 行以下のプログラムにしては，悪くはないでしょう。

TMyApp は，ウィンドウを初期化すると，デスクトップにそのウィンドウを挿入します。以前に説明したことですが，***TDeskTop*** はグループです。グループですから，ウィンドウと同様，サブビューを持ち，管理することができます。プログラムをコンパイルして，実際に実行させてみればわかることですが，新しくオープンしたウィンドウをリサイズしたり，移動させたり，クローズしたりできます。マウスからの入力は，一連のイベントに変換され，デスクトップからその新しいウィンドウに伝えられます。そのウィンドウが，それらのイベントを処理するのです。

cmNewWin を発生させつづけると，デスクトップ上にどんどんウィンドウが表示されていきます。ウィンドウごとに番号が付けられます。これらのウィンドウは，お互い，他のウィンドウ上でリサイズしたり，選択したり，移動させたりすることができます。**図 2.2**

は，デスクトップ上でウィンドウがいくつもオープンしている様子を示しています。

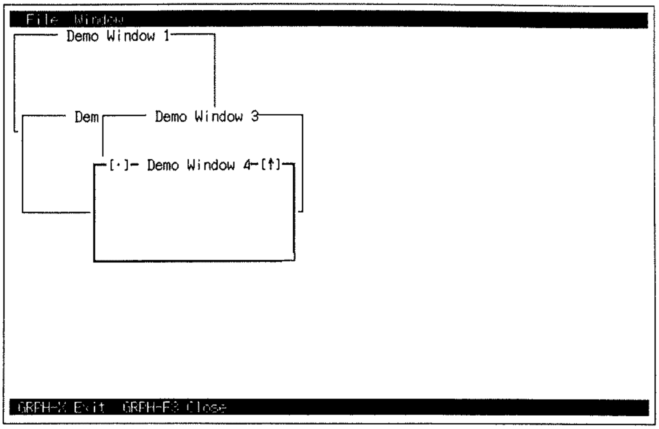


図 2.2 TVGUID04 を実行して，複数のウィンドウをオープンしている様子

TWindow オブジェクトは，グループですが，最初は，ビューとして ***TFrame*** オブジェクトしか備えていません。ユーザがウィンドウを移動させたり，リサイズしたり，クローズしたりするには，フレームアイコン上をクリックします。フレームは，ウィンドウの初期化の際にもらったタイトルを表示します。またフレームは，***TBackGround*** がデスクトップに対して行なうように，ウィンドウの背景を描きます。これらのことは，先に示したように，何も書かなくても実行されます。

ウィンドウの内部

従来のウィンドウを扱うのであれば，次にウィンドウ内に何かを書くことになります。しかしながら，***TWindow*** は，その上に何かが書かれる真っ白な石板というわけではありません。***TWindow*** は，***TGroup*** クラスであり，フレームビュー以外には，画面には何も表示されません。何かをウィンドウ“内に”表示させるには，もう一歩進まなければならないのですが，この一歩がものすごい力を発揮する一歩なのです。

何かをウィンドウ内に表示する場合，まず自分自身を描いて，そのあとウィンドウ内に挿入する機能を持つビューを作成することに

なります。こういうビューのことを**インテリア**といいます。

最初にとりあげるインテリアは，ウィンドウ全体をフィルするものです。もちろん，後になれば，そのインテリアを小さくしたり，内部に他のビューを描くことも簡単にできることがわかります。ウィンドウは，複数のインテリアを持つことができ，またそれ以外にも，入力行や，ラベル，ボタン，チェックボックスといった便利なビューを，いくつでも持つことができます。ウィンドウのフレームに，スクロールバービューを持たせることも簡単にできます。

グループ内では，サブビューをタイル貼りにしたり，カスケード表示したり，オーバーラップさせたりすることができます。ビュー同士がどう関係するかは，プログラムしだいです。***TDeskTop*** には，***tile*** と ***cascade*** というメンバ関数が用意されており，サブビューを初期化した後，タイル貼りにしたり，カスケード表示したりします。ただしこれらのメンバ関数は，デスクトップに関してしか使うことができません。

次に作成するインテリアは，単なる ***TView*** からの導出クラスです。すべての ***TView*** が，従来の静的なウィンドウの場合と同じ働きのフレームを備えることができます。静的というのは，マウスのクリックに応答しないということです。***TView*** のフレームは，クリッピング領域の外側に置かれるものです。つまり，このフレームは，ビューを囲む単なる線だということです。

TView インテリアが，オーナーウィンドウ全体をフィルするのであれば，そのインテリアがフレームを持とうが持つまいが関係ありません。ウィンドウのフレームが，インテリアのフレームをおおってしまうからです。インテリアがウィンドウよりも小さい場合には，インテリアのフレームは見えることになります。その場合，後でそういう例がでてきますが，ウィンドウ内の複数のインテリアにフレームが描かれることになります。

以下のリストは，デモウィンドウ内に“Hello, World!”と表示するプログラムで，**図 2.3** がその実行結果です。

これは，TVGUID05.CPP の一部です。

```
#include <stdlib.h>
// random() のため

class TInterior : public TView
{

public:
```

```
    TInterior(TRect bounds);    // コンストラクタ
    virtual void draw();        // TView::draw をオーバーライド
}

TInterior::TInterior(const TRect& bounds) : TView(bounds)
{
    growMode = gfGrowHiX | gfGrowHiY;    // サイズをウィンドウに
    options = options | ofFramed;        // 合わせる
}

void TInterior::draw()
{
    char *hstr = "Hello World!";
    ushort color = getColor(0x0301);
    TView::draw();
    TDrawBuffer b;
    b.moveStr( 0, hstr, color );
    writeLine( 4, 2, 12, 1, b);
}

TDemoWindow::TDemoWindow(const TRect& bounds, const char *aTitle,
               short aNumber) : TWindow( bounds, aTitle,
               aNumber), TWindowInit( &TDemoWindow::initFrame )

{
    TRect r = getClipRect();  // 露出した領域を取得
    r.grow(-1,-1); // インテリアがウィンドウフレーム内に入るように縮小
    insert( new TInterior( r )); // ウィンドウにインテリアを追加
}
```

このプログラム例で文字列ストリームを使用する場合には，#include 〈strstrea.h〉を追加する必要があります。

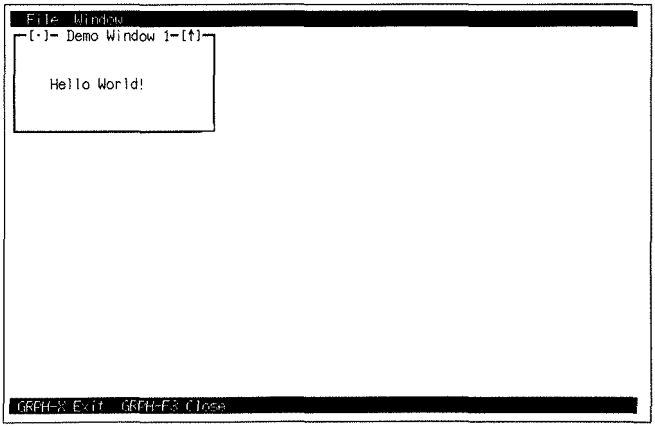


図 2.3
TVGUID05 を実行させて，ウィンドウがオープンされた様子

何が見える？

ビューは，すべて，自分自身を描く機能を備えています。ビューは，メンバ関数 ***draw*** を用いて描かれます。新しいビューを導出して何か新しい表示を行ないたい場合には，基本クラスの ***draw*** メンバ関数をオーバーライドして，画面に自分自身を表示させるようにしなければなりません。*TInterior* は，***TView*** から導出されたクラスであり，新しい ***draw*** メンバ関数を定義する必要があります。

新しく定義した *TInterior::draw* は，まず，基本クラス ***TView*** の ***draw*** を呼び出します。基本クラスの ***draw*** は，ここでは，ビューの矩形領域をクリアするだけです。普通はこういうことは行ないません。新たに作るインテリアビューの ***draw*** メンバ関数は，領域全体の面倒をみるのが普通であり，したがって ***TView::draw*** 呼び出しは意味がないからです。

ウィンドウのインテリア内に，本当に何か描きたいのであれば，やはり，継承した ***draw*** メンバ関数を呼び出すということはしないでしょう。***TView::draw*** を呼び出すと，インテリアに2回以上描かれるところがでてきて，表示がどたばたしてしまうからです。

試しに，TVGUID05.CPP から ***TView::draw*** 呼び出しの部分をコメントにして削除し，再コンパイルしてみるとよいでしょう。そうして，ウィンドウを移動させたり，リサイズしてみてください。そうすれば，なぜビューが自分の領域全体の面倒をみなければならないかが，よくわかると思います。

☞ Turbo Vision は，ユーザがビューをオープンしたり，クローズしたり，移動させたり，リサイズした場合には，いつでもビューの ***draw*** メンバ関数を呼び出します。しかし，ビューに自分自身を再表示させたいのであれば，***draw*** ではなく，***drawView*** を呼び出してください。***drawView*** は，ビューのうち画面上で見えるものだけを表示します。次のことが肝要です。***draw*** はオーバーライドしてもよいが，決して直接呼び出さないこと，***drawView*** は呼び出してもよいが，決してオーバーライドしないこと！

write の扱い方

使おうと思えば，Turbo Vision で **printf** や **puts** あるいは C++ 標準のストリーム入出力を使用することも可能ですが，処理内容によっては，使用する道具としてふさわしくない場合があります。第一に，単に何かを書き込む場合，ウィンドウやその他のビューがそれとダブったり消されたりしないようにすることが不可能だということです。第二に，現在のビューのローカル座標に書き込む必要があり，そのビューの境界でクリップしなければならないということです。第三に，書き込みを行う場合，どの色を使うかという問題があるということです。また ***TView*** は，TVGUID05 で使用されている ***writeLine*** などの専用の表示ツールを用意しています。たとえば，***TDrawBuffer*** オブジェクトを使用するツールもあれば，引数に直接文字列や文字をとるツールもあります。表示バッファについては TVGUID07 を参考にしてください。

TView::writeStr には，ローカル座標を使って書き込んだり，親のビューの境界内のビューをクリップするなどという機能はありませんし，またビューのカラーパレットの使い方についても関知していません。***writeStr*** は，引数として，x 座標，y 座標，出力する文字列，およびカラーインデックスをとります。

```
void writeStr( short x, short y, const char *str,
               uchar color);
```

writeChar は，***writeStr*** と似ており，***TView*** で以下のように宣言されています。

```
void writeChar( short x, short y, ushort ch, uchar color,
                short count);
```

writeStr と同様，***writeChar*** は，出力をビュー内の *x* 座標，*y* 座標から開始し，文字 *ch* を *count* 回だけ書き出します。これらの関数はともに，ビューパレット内の *color* 番目のエントリで指定される色を使用して表示します。

これらの ***writeXXX*** メンバ関数は，それぞれビューの ***draw*** メンバ関数内からしか呼び出してはなりません。Turbo Vision においては，何か書き込みを行う必要があるのは，***draw*** メンバ関数内でだけです。

簡単なファイルビュアー

この節では，ウィンドウに新しい機能を追加し，インテリア内に実際に何か表示させます。ディスクからテキストファイルを読み出し，それをインテリア内に表示するメンバ関数を加えます。

注意！ このプログラムでは，文字がいくらか“ゴミ”になって出てきます。でも心配する必要はありません。意図的にそうしたのです！

これは，TVGUID06.CPP です。

```
const char *fileToRead = "tvguid06.cpp";
const int maxLineLength = maxViewWidth + 1;
const int maxLines = 100;
char *lines[maxLines];
int lineCount = 0;

void readFile( .const char *fileName )
{
 ...
// fileName から文字列配列 lines に読み込む
 ...
}

void TInterior::draw()
{
    for( int i = 0; i < size.y; i++ )
        writeCStr( 0, i, lines[i], 1 );
}

int main()
{
    readFile( fileToRead );
    TMyApp myApp;
    myApp.run();
    deleteFile();        // 文字列配列 lines を削除
    return 0;
}
```

テキストファイルの読み出し

アプリケーションでは，*readFile* を呼び出して，テキストファイルを *lines* 配列にロードします。

表示のバッファリング

このプログラムを走らせてみると，空白行になるべきところに，文字の“ゴミ”がでてくることがわかります。これは，***draw*** メンバ関数がちゃんとした仕事をしていないためです。各ビューの ***draw*** メンバ関数は，ビューが面倒をみるべき領域全体をサポートしなければならない，という原則が守られていないのです。

また，テキスト配列 ***lines*** も，ビュー内にしかるべき形で表示されてはいません。テキストは，通常，可変長の文字列からなり，長さ0の文字列のものもかなりあります。***draw*** メンバ関数は，インテリアの領域全体をサポートする必要があるので，各テキスト行は，ビューの横幅に合わせて，1行を長くしたりする必要があります。

表示バッファ

これをサポートするために新しい ***draw*** を作成し，ウィンドウに表示する前に，1行ずつバッファ内で組み立てを行なうようにします。***TDrawBuffer*** クラスは，表示バッファを操作するためのメンバ関数として ***moveXXX*** や ***putXXX*** を用意しています。***TView*** は，これに対応して表示バッファの内容を画面に表示する（書き込む）ための関数 ***writeXXX*** を用意しています。

TDrawBuffer オブジェクトは，属性バイトと文字バイトを対にして格納します。***TView::writeBuf*** を使用して画面に表示バッファの内容を書き込むことができます。***TView::writeBuf*** は，VIEWS.H で以下のように宣言されています。

```
void writeBuf(short x, short y, short w, short h,
              const void far *b);
void writeBuf(short x, short y, short w, short h,
              const TDrawBuffer& b);
```

これらの関数は，*b* で指定されるバッファの内容を，座標 *(x, y)* から始まる幅 *w*，高さ *h* の画面領域に書き込みます。1番目の形式の関数は，文字および属性からなるダブルワード配列（下位ワードが表示コード，上位ワードが表示属性）を表示します。2番目の形式

の関数は，***TDrawBuffer*** のインスタンスを表示します。なお ***writebuf*** は，***draw*** メンバ関数内からしか呼び出すことはできません。

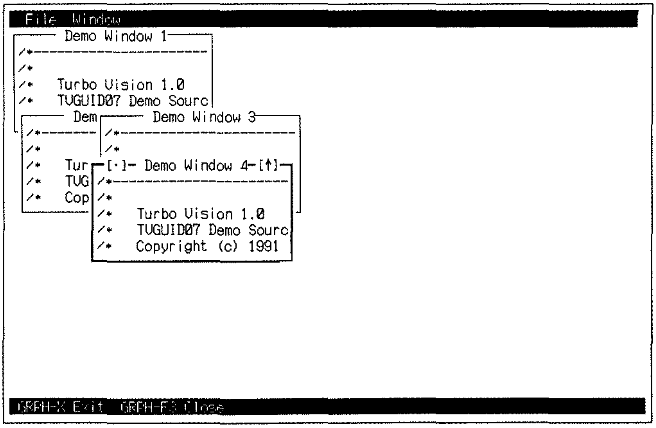
新しい *TInterior::draw* は，以下のとおりです。

これは，TVGUID07.CPP です。

```
void TInterior::draw()
{
    ushort color = getColor(0x0301);
    for( int i = 0; i < size.y; i++ )
        {
        TDrawBuffer b;
        b.moveChar( 0, ' ', color, size.x );
        // ラインバッファをスペースで埋める
        if( lines[i] )
            {
            char s[maxLineLength];
            strncpy( s, lines[i], size.x );
            s[size.x] = EOS;
            b.moveStr( 0, s, color );
            }
        writeLine( 0, i, size.x, 1, b);
        }
}
```

図2.4 に，TVGUID07を実行して，ウィンドウをいくつかオープンした様子を示します。

図2.4
複数のファイルビュー



draw は，まず ***TDrawBuf::moveChar*** を呼び出して，*size.x* 個のスペース（インテリアの幅分）を，色もちゃんと指定して ***TDrawBuffer*** オブジェクトの ***b*** に書き込みます。これで各行は，インテリアの横幅に合わせてスペースで長さが揃えられたことになります。次に ***b.moveStr*** を呼び出してテキスト行を ***b*** にコピーし，そこで

writeLine を呼び出してバッファ全体を表示します。

テキストをバッファに移す

TDrawBuffer は，テキストを ***TDrawBuffer*** に転送するために ***moveStr***，***moveSubStr***，***moveChar***，***moveCStr*** および ***moveBuf*** の 5 個のメンバ関数を用意しています。これらのメンバ関数は，それぞれ文字列，部分文字列，文字，制御文字列（メニュー項目やステータス項目で使用されるチルダ("~")を含めた文字列）および別のバッファ内容を，呼び出し側のバッファに転送します。これらの関数の詳細については第 13 章に示してあります。

バッファの内容の表示

writeXXX の各関数については，すべて第 13 章に詳細を示してあります。

TView は，固定文字列やバッファの内容をビューに出力するために 4 個のメンバ関数を用意しています。その中の ***writeBuf*** と ***writeLine*** の 2 つは，***TDrawBuffer*** 型の引数をとります (残りの 2 つは ***writeChar*** と ***writeStr***)。この 4 個のメンバ関数はすべて，バッファを使用しません。またいずれも ***draw*** メンバ関数内からしか呼び出してはなりません。***writeBuf***，および ***writeStr*** については，すでに示してあるので，残りの ***writeLine*** と ***writeCStr*** についてプロトタイプを示しておきます。

```
void writeLine(short x, short y, short w, short h,
               const TDrawBuffer& b);
void writeLine(short x, short y, short w, short h,
               const void far *b);
void writeCStr(short x, short y, char far *str, uchar color);
```

TInterior::draw では，***writeLine*** で ***TDrawBuffer*** オブジェクトの内容を 1 行として出力しますが，4 番目のパラメータ *h*(行数) に 1 以上を指定すれば，バッファの内容をその数だけ繰り返して出力します。たとえば，*buf* に “Hello, World!” と書かれていたとすると，writeLine(0,0,13,4,buf) によって，

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

と表示されます。

writeBuf (*x, y, w, h, buf*) も，画面の矩形領域に出力を行ないますが，*w* と *h* で，バッファの横幅と行数が指定できます。たとえば，*buf* の内容が "ABCDEFGHIJKLMNOP" だったとすると，writeBuf (0,0,4,4,buf) によって，

```
ABCD
EFGH
IJKL
MNOP
```

と表示されます。

writeStr および ***writeChar*** の場合とは異なり，***writeBuf*** および ***writeLine*** は，引数にカラーパレットをとりません。これは，テキストをバッファに転送する場合に色の指定が行なわれるからです。つまり1つのバッファにいろいろな属性を持つテキストを入れることができるためです。

何行表示するかを知る

TInterior.draw は，インテリアがちょうど一杯になる分だけしかファイルの内容を表示しませんでした。そうせずに，ファイルのうち *TInterior* の境界でクリップされるところまでの部分を出力するようにしたとすると，***draw*** は，ほとんどそれにかかりっきりになってしまいます。

ビューの表示にすごく時間がかかるようであれば，先に ***getClipRect*** を呼び出せばよいでしょう。***getClipRect*** の宣言は以下のとおりです。

```
TRect TView::getClipRect();
```

getClipRect は，そのビューの中の表に現われている矩形領域を返します。したがって，その部分だけを表示すればよいわけです。たとえばユーザが，いろいろなサブビューで構成されているダイアログボックスを移動させて，それまで隠されていたものを見えるようにした場合，表示する前に ***getClipRect*** を呼び出せば，一時的に画面に表示されなくなった部分は再表示する必要がなくなります。***getClipRect*** と ***getExtent*** を混同しないようにしてください。

getExtent は，ビューの領域の中どれだけが表示されているかには関係なく，現在の境界を返します。

スクロールアップ/スクロールダウン

ファイルの最初の数行しか見ることができないのだったら，ファイルビュアーは，ほとんど役に立ちません。そこで今度は，インテリアをスクロールビューに変え，スクロールバーを持たせることにします。*TInterior* は，テキストファイルをスクロールすることのできるウィンドウになるわけです。また，*TDemoWindow* にも変更を加えます。*makeInterior* メンバ関数を追加して，ウィンドウをオープンする機能とは別個の機能を持たせます。

これは，TVGUID08.CPP です。

```
class TDemoWindow : public TWindow // 新しいウィンドウクラスを定義
{
public:

    TDemoWindow( const TRect& bounds, const char *aTitle,
                 short aNumber );
    void makeInterior();

};
```

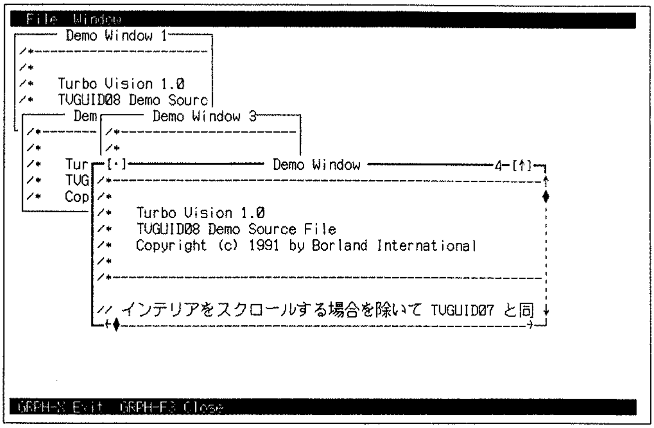
TInterior の基本クラスを変更していることに注意してください。

```
class TInterior : public TScroller
{

public:

    TInterior( const TRect& bounds, TScrollBar *aHScrollBar,
               TScrollBar *aVScrollBar ); // コンストラクタ
    virtual void draw();      // TView::draw をオーバーライド
};

// TInterior の定義

TInterior::TInterior( const TRect& bounds, TScrollBar
                *aHScrollBar, TScrollBar *aVScrollBar ) :
       TScroller( bounds, aHScrollBar, aVScrollBar ) {
    growMode = gfGrowHiX | gfGrowHiY;
    options = options | ofFramed;
    setLimit( maxLineLength, maxLines );
}

void TInterior::draw()        // スクローラ用に変更
{
    ushort color = getColor(0x0301);
    for( int i = 0; i < size.y; i++ )
```

```
        // 1行ごとにループ
        {
        TDrawBuffer b;
        b.moveChar( 0, ' ', color, size.x );
        // ラインバッファをスペースで埋める
        int j = delta.y + i;  // delta はスクローラオフセット
        if( lines[j] )
            {
            char s[maxLineLength];
            if( delta.x > strlen(lines[j] ) )
                s[0] = EOS;
            else
                {
                strncpy( s, lines[j]+delta.x, size.x );
                s[size.x] = EOS;
                }
            b.moveStr( 0, s, color );
            }
        writeLine( 0, i, size.x, 1, b);
        }
}

// TDemoWindow definitions
void TDemoWindow::makeInterior()
{
    TScrollBar *vScrollBar =
        standardScrollBar( sbVertical | sbHandleKeyboard );
    TScrollBar *hScrollBar =
        standardScrollBar( sbHorizontal |  sbHandleKeyboard );
    TRect r = getClipRect(); // 露出するビュー境界を取得
    r.grow( -1, -1 );     // ウィンドウフレーム内に入るように縮小
    insert( new TInterior( r, hScrollBar, vScrollBar )); }

TDemoWindow::TDemoWindow( const TRect& bounds,
             const char *aTitle, short aNumber) :
         TWindow( bounds, aTitle, aNumber),
         TWindowInit( &TDemoWindow::initFrame )
{
    makeInterior(); // スクロール可能なインテリアを作成して
                    // ウィンドウに挿入
}
```

図 2.5
スクロールするインテリアを備えたファイルビュアー

水平スクロールバーと垂直スクロールバーは，初期化されグループに挿入された後，TScroller の初期化時に TScroller に渡されます。



スクローラとは，それよりも大きな仮想ビューの一部を表示するためのビューのことです。スクローラおよびそのスクロールバーによって，スクロール可能なビューが作り出されます。その際，プログラマは，ほとんど何もする必要はありません。プログラマがやらなければならないことは，スクローラ用の ***draw*** メンバ関数を作成して，スクローラが仮想ビューの中のしかるべき部分を表示できるようにすることだけです。スクロールバーが，自動的にスクローラの ***delta.x***（表示を開始する桁位置）および ***delta.y***（表示を開始する最初の行）の値を制御します。

役に立つスクローラにするには，***TScroller*** の ***draw*** メンバ関数をオーバーライドする必要があります。スクロールバーに応じて，***delta*** の値は変化しますが，***delta*** が何かを表示するというわけではありません。***delta*** が変化すると，必ず ***draw*** メンバ関数が呼び出されますので，そのとき ***delta*** を参照して処理を行なうように記述することになります。

1 個のウィンドウに複数のビューを持たせる

次に，インテリアの複製を作り，スクロールを行なうビューを 2 個備えたテキストファイル用のウィンドウを作ります。マウスもしくはタブキー入力で，自動的に，2 つあるインテリアビューのうちの 1 つが選択されます。それぞれのビューは独立してスクロールさ

せることができ，それぞれにカーソル位置を持たせます。

これを実現するためには，*makeInterior* メンバ関数にセクションを1個設け，ウィンドウのどちらの側にインテリアがあるか（どちらにあるかで動作が少し異なる）がわかるようにします。*TDemoWindow* のコンストラクタからは *makeInterior* を2度呼び出すようにします。

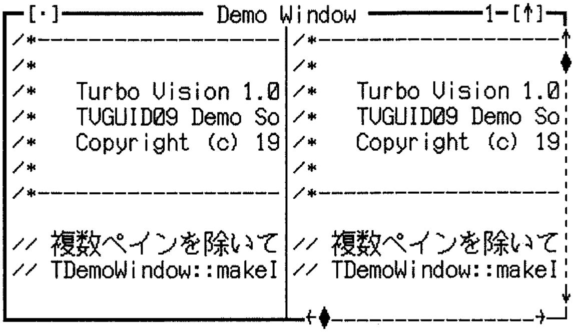
これは，TVGUID09.CPP です。

☞ **makeInterior の宣言も変更することを忘れないように。**

```
// new TDemoWindow constructor
TDemoWindow::TDemoWindow( const TRect& bounds,
              const char *aTitle, short aNumber) :
         TWindow( bounds, aTitle, aNumber),
         TWindowInit( &TDemoWindow::initFrame )
{
  bounds = getExtent();
  TRect r( bounds.a.x, bounds.a.y, bounds.b.x/2+1,
           bounds.b.y );
  TInterior *lInterior = makeInterior( r, True );
  lInterior->growMode = gfGrowHiY;
  insert( lInterior );
  // 左側がスクロール可能なインテリアを作成してウィンドウに挿入
  r = TRect( bounds.b.x/2, bounds.a.y, bounds.b.x, bounds.b.y
  TInterior *rInterior = makeInterior( r, False );
  rInterior->growMode = gfGrowHiX | gfGrowHiY;
  insert( rInterior );
  // 右側のスクローラも同様
}

// 新しいメンバ関数 makeInterior
TInterior *TDemoWindow::makeInterior( const TRect& bounds,
           Boolean left )
{
  TRect r = TRect( bounds.b.x-1, bounds.a.y+1, bounds.b.x,
                   bounds.b.y-1 );
  TScrollBar *vScrollBar = new TScrollBar( r );
  if( vScrollBar == 0 )
     {
     cout << "vScrollbar init error" << endl;
     exit(1);
     }
     // 生成コードはエラーダイアログボックスを表示する
  vScrollBar->options |= ofPostProcess;
  if( left )
     vScrollBar->growMode = gfGrowHiY;
  insert( vScrollBar );

  r = TRect( bounds.a.x+2, bounds.b.y-1, bounds.b.x-2,
             bounds.b.y );
  TScrollBar *hScrollBar = new TScrollBar( r );
  if( hScrollBar == 0 )
     {
     cout << "hScrollbar init error" << endl;
     exit(1);
```

```
   }
hScrollBar->options |= ofPostProcess;
if( left )
   hScrollBar->growMode = (gfGrowHiY | gfGrowLoY);
insert( hScrollBar );

r = bounds;
r.grow( -1, -1 );
return new TInterior( r, hScrollBar, vScrollBar ); }
```

図 2.6
複数の枠を持つ
ウィンドウ



TVGUID09.CPP を実行させて，ウィンドウを小さくした場合，ウィンドウの右側のビューを左側のビューに近づけすぎると，垂直方向のスクロールバーの上に左のインテリアが重ね書きされることがわかります。これを避けるには，ウィンドウをどれぐらいまで小さくできるかについて制限を設ければよいでしょう。このためには，***TWindow*** の仮想メンバ関数 ***sizeLimits*** をオーバーライドします。

これは，TVGUID10.CPP です。
TDemoWindow に sizeLimits 宣言を追加することを忘れないでください。

```
void TDemoWindow::sizeLimits( TPoint& minP, TPoint& maxP ) {
    TWindow::sizeLimits( minP, maxP );
    minP.x = lInterior->size.x+9;
}
```

TDemo Window::sizeLimits を呼び出す必要はありません。単にオーバーライドすればよいのです。そうすれば，***sizeLimits*** は必要なときに呼び出されます。これは，***draw*** メンバ関数で行なったことと同じことです。ビューに，そのビュー自身を**表示させるようにはする**けれども，**いつ**表示させるかについては記述しません。Turbo Vision は，いつ ***draw*** を呼び出せばよいかを知っていたわけです。***sizeLimits*** についても同様です。ウィンドウの大きさの制限を設けておけば，ビューが適当なときにそのチェックを行なってくれるわけです。***sizeLimits*** は仮想のメンバ関数なので，いつでも，ビュー

の種類に応じて適当な関数が呼び出されます。

どこにどういう機能を持たせるか

上では，1個のウィンドウにビューをたくさん持たせました。フレームを1個，スクロールするインテリアを2個，それに各インテリアにスクロールバーが2個ずつ，でした。次に，アプリケーションの一部として，ある特定の機能を果たすウィンドウを作ることにしましょう。

どういう機能にしましょうか。たとえば，ウィンドウに本格的なテキストエディタの機能を持たせることにしましょう。ウィンドウには，ビューが2個あるので，グループにテキスト編集の機能をいくらか持たせ，グループにその2個のビューと通信させるようにすればよいでしょう。つまり，ビューの管理がグループの仕事になります。グループには作業全体のことに専念させるというのが，自然な考え方なのではないでしょうか。

グループは，もちろんビューと同じく，どんどん拡張することができ，どんな機能でも持たせることができるのですが，アプリケーションをより一層完璧かつ柔軟なものにするためには，次の2つの方針を守っておくほうがよいでしょう。すなわち，**クラスにはできるだけ自律性を持たせること**，さらに**グループ（ウィンドウなど）にはできるだけ余計な機能を持たせないようにすること**，この2点です。

これに従うとすれば，テキストエディタを作成する場合，その機能はすべてインテリアビューに持たせたほうがよいことになります。つまり，テキストエディタのビュー型を作成するわけです。ビューの設計をうまくやりさえすれば，ビューを再利用するのは簡単です。下手に編集機能をグループとビューとに分けて持たせたりすると，テキストエディタを他の環境で再利用するのが難しくなります。

ダイアログボックスの作成

使用するクラス
TView
TGroup
TDialog
TCluster
TCheckBoxes
TRadioButtons
TLabel
TInputLine

ダイアログボックスは，ウィンドウの一種です。実際，***TDialog*** は，***TWindow*** の導出クラスです。ただし，ウィンドウとして扱うことは**できます**が，通常は，ウィンドウとは少し異なることをやらせます。

例題プログラムでダイアログボックスを使うには，まず，ダイアログボックスをオープンするためのメニュー項目を新たに設け，実際にオープン処理を行なうメンバ関数をアプリケーションに付加します。そして，メニュー項目が発行したコマンドとその実際の処理とをリンクするための1行を，アプリケーションの ***handleEvent*** メンバ関数に書き加えます。

TWindow の場合には，新しいクラスを導出しましたが（*TDemoWindow* を作成した），***TDialog*** の場合には，そうする必要はありません。ダイアログボックスの特殊な型を作り出すかわりに，アプリケーションにいろいろな機能を付加します。つまり，必要な機能を装備するダイアログボックスのクラスを新たにインスタンス化するのではなく，汎用のダイアログボックスをそのままインスタンス化して，それに必要な仕事を**やってもらう**のです。

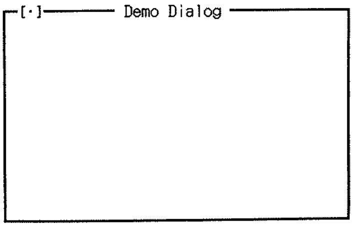
ダイアログボックス同士で異なる点は，それぞれのダイアログボックスに何が付いているかということだけであり，ダイアログボックス自体の機能は異ならないので，***TDialog*** からクラスを導出する必要は，まずありません。

これは，TVGUID11.CPP です。

```
// ダイアログメニュー用に追加
const int cmNewDialog  = 202;

TMenuBar *TMyApp::initMenuBar( TRect r )
{
  r.b.y = r.a.y + 1;     // 最下行を最上行の1行下に設定
  return new TMenuBar( r, *new TSubMenu( "~F~ile", kbAltF )+
    *new TMenuItem( "~O~pen", cmMyFileOpen, kbF3, hcNoContext,
                    "F3" )+
    *new TMenuItem( "~N~ew", cmMyNewWin, kbF4, hcNoContext,
                    "F4" )+
    newLine()+
    *new TMenuItem( "E~x~it", cmQuit, cmQuit, hcNoContext,
                    "Grph-X" )+
```

```
        *new TSubMenu( "~W~indow", kbAltW )+
        *new TMenuItem( "~N~ext", cmNext, kbF6, hcNoContext, "F6" )+
        *new TMenuItem( "~Z~oom", cmZoom, kbF5, hcNoContext, "F5" )+
        *new TMenuItem( "~D~ialog", cmNewDialog, kbF2, hcNoContext,
                        "F2" )
        // ここに新しいダイアログメニューを追加する
      );
}

// TMyApp にはメンバ newDialog が必要
void TMyApp::newDialog()
{
    TRect r( 0, 0, 40, 13 );
    r.move( random(39), random(10) );
    deskTop->insert( new TDialog( r, "Demo Dialog" )); }

// cmNewDialog を処理できるように handleEvent を変更
void TMyApp::handleEvent(TEvent& event)
{
    TApplication::handleEvent(event);
    if( event.what == evCommand )
        {
        switch( event.message.command )
            {
            case cmMyNewWin:
                newWindow();
                break;
            case cmNewDialog:
                newDialog();
                break;
            default:
                return;
            }
        clearEvent( event );    // イベントを処理したらクリア
        }
}
```

図 2.7
簡単な
ダイアログボックス



このダイアログボックスと以前作成したウィンドウとで異なるのは以下の点だけです。

■ダイアログボックスのデフォルトの色は，青ではなく，灰色(拡張アトリビュートモード時）である。

■ダイアログボックスの場合，リサイズやズームはできない。
■ダイアログボックスは，ウィンドウ番号を持たない。

ダイアログボックスをクローズするには，ダイアログボックス上のクローズアイコンをクリックするか，ステータスラインの項目GRPH-F3をクリックします。何も指定しなければ，**ESC** キーは，ダイアログボックスをキャンセルする機能を持ちます。

上のダイアログボックスは，モードなしダイアログボックスと呼ばれるものです。ダイアログボックスには通常，モードがあります。つまり，ダイアログボックスが操作モードを決めます。通常，ダイアログボックスをオープンすると，そのときにアクティブなのは，ダイアログボックスだけです。そういうのが**モードつきビュー**です。ダイアログボックスのモードになっているかぎり，他のウィンドウやメニューをクリックしても，何も起こりません。モードなしダイアログボックスを使いたい場合もありますが，ダイアログボックスをモードつきにしたい場合が圧倒的に多いでしょう。

モードつきビューについては第4章『ビュー』で説明します。

モードつきダイアログボックスを実行する

それでは，ダイアログボックスをモードつきにするにはどうすればよいのでしょうか。それはいとも簡単なことです。ダイアログボックスのクラスをデスクトップに**挿入する**かわりに，***deskTop->execView*** 関数を呼び出すことで，それを**実行すれば**よいのです。

これは，TVGUID12.CPP です。

```
// tvguid11 から変更: こんどは execView を呼び出す
void TMyApp::newDialog()
{
   TRect r( 0, 0, 40, 13 );
   r.move( random(39), random(10) );
   deskTop->execView( new TDialog( r, "Demo Dialog" )); }
```

TDialog オブジェクト自体が，**ESC** キーイベントおよび**リターン**キーを処理します（**ESC** キーは，***cmCancel*** コマンドに変換され，**リターン**キーは，ダイアログボックスのデフォルトの ***TButton*** によって処理されます)。ダイアログボックスは，常に ***cmCancel*** コマンドでクローズされます。

execView を呼び出すことで，ダイアログボックスはグループに

挿入され，モードつきになります。ダイアログボックスがクローズするかキャンセルされると，***execView*** はグループからダイアログボックスを取り除き，終了します。それまでは，***execView*** はダイアログボックス内に残っています。***execView*** 関数からの戻り値は *control* に格納されますが，当面は，この値は無視することにします。TVGUID16でこの値を利用します。

コントロールの設置

当然ながら，何も持たないダイアログボックスは，ほとんどダイアログボックスの役をなしません。ダイアログボックスに何か働きをさせるためには，**コントロール**を付加する必要があります。コントロールとは，ダイアログボックス中にあって情報の処理を可能にするいろいろな要素のことです。コントロールで重要なことは，それがダイアログボックス内のものについてのみ影響を及ぼすということです。

コマンド処理については，第5章『イベント駆動型プログラミング』のところで詳細に説明します。

このルールに対する唯一の例外が，モードなしダイアログにおけるボタンの場合です。ボタンは，コマンドを発生するので，現在のモードつきビューから下向きにそのコマンドがどんどん伝播していきます。ダイアログボックスがモードつきビューでない場合には，コマンドはダイアログボックス外に伝えられることになり，思わぬ結果を招くことになります。

一般に，ダイアログボックスにコントロールを設定する場合，画面上の表示とデータの処理とを分離して考えることができます。つまりダイアログボックス自体は，そこで扱われるデータを設定したり使用したりするプログラムを書かなくても，簡単に設計できるということです。これは，ちょうどメニューやステータスの項目が，発生コマンドの処理プログラムを作成しなくても設定できたのと同じことです。

ボタン，ボタン，...

コントロールクラスの中でも，もっとも簡単なものの1つが ***TButton*** です。これは，ちょうどステータスラインを飾る項目と同じ働きをするもので，テキストラベルを持つ色付きの領域のことです。その領域の上をクリックすると，コマンドを発生します。また，ボタンの“背後”には陰があり，ボタンをクリックすると，3次元的な表示を行ないます。

たいていのダイアログボックスは，少なくともボタンを1個ないしは2個持ちます。もっとも一般的なボタンが“OK”(これで大丈夫です。ダイアログボックスをクローズして，この設定内容を使ってください，という意味)および“Cancel”(ダイアログボックスをクローズし，ここで行なった設定変更はすべて無視してください，という意味)です。Cancelボタンは，通常，クローズアイコンが発生するのと同じ ***cmCancel*** コマンドを発生します。

TButton にリンクすることのできる標準コマンドが5個あります。***cmOk***，***cmCancel***，***cmYes***，***cmNo***，および ***cmDefault*** です。最初の4個のコマンドは，***TDialog*** に ***endModal*** メンバ関数を呼び出させて，ダイアログボックスのクローズも行ないます。このメソッドは，モードつきステータスを，それ以前のモードつきビューのものに戻します。

これらのコマンド以外にも，アプリケーションに固有なコマンドを発生するボタンを使うことができます。

これは，TVGUID13.CPPです。

```
// tvguid12 から変更: ボタンを追加
void TMyApp::newDialog()
{
  TDialog *pd = new TDialog(TRect(20, 6, 60, 19),"Demo Dialog");
  if( pd )
  {
    pd->insert( new TButton( TRect(15, 10, 25, 12), "~O~K",
                cmOk, bfDefault ));
    pd->insert( new TButton( TRect(28, 10, 38, 12), "~C~ancel",
                cmCancel, bfNormal ));
    deskTop->execView( pd );
  }
  destroy( pd );
}
```

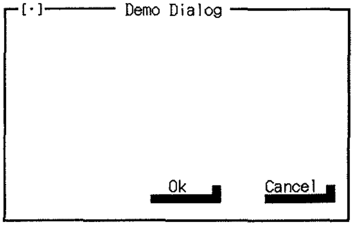
ボタンを作成するには，コンストラクタに以下の4つの引数を指

定する必要があります。

1. ボタンがサポートする領域（陰の部分を含めることも忘れないでください）
2. ボタンに表示するテキスト
3. ボタンにリンクするコマンド
4. ボタンの種類を示すフラグ（ノーマルもしくはデフォルト）

ユーザがダイアログボックスをクローズしたりキャンセルしたりした場合，制御は ***execView*** から離れ，ダイアログボックスは消去されます。***new*** を使用したら，通常なら C++ の delete を使用するところですが，かわりに ***destroy*** を使用する必要があることに注意してください。***TObject*** の導出クラスで ***new*** を使用して作成されたオブジェクトは，すべて ***TObject*** から継承した ***destroy*** メンバ関数を使用して廃棄する必要があります。delete の場合とまったく同様，引数にオブジェクトへのポインタを 1 個とります。このようにする理由を知りたい方は，第 6 章の Turbo Vision のメモリ管理に関する説明，および第 13 章（***TVMemMgr*** クラスと ***TObject*** クラス）をお読みください。とりあえずは，***TObject*** からの導出オブジェクトを廃棄する場合に必要な扱いの難しい内部的な処理は，すべて ***destroy*** が面倒をみてくれ，消去したオブジェクトと関連のあるすべてのオブジェクトの状態も適切に再設定してくれる，ということを理解していただければ充分です。

図 2.8
ダイアログボックスのボタン



ダイアログボックスをキャンセルするためのホットキー（***ESC***）が決まっていますので，“Cancel” の “C” はハイライトしませんでした。こうすれば，***C*** をそれ以外のコントロール用のショートカットに使用できます。

ノーマルボタンとデフォルトボタン

ボタンには，フラグとして ***bfNormal*** あるいは ***bfDefault*** を持たせる必要があります。普通は，***bfNormal*** です。フラグが ***bfDefault*** のボタンは，デフォルトのボタンであり，**リターン**キーを押せば，そのボタンが“押された”ことになります。Turbo Visionは，デフォルトボタンが1個しかないかどうかはチェックしません。開発する側でそのように設計しなければなりません。デフォルトコントロールを2個以上設けると，結果は予測できません。

通常は，“OK”ボタンをデフォルトボタンとします。これによって，ユーザは**リターン**キーを押せば，ダイアログボックスをクローズし，設定内容を更新できることに慣れていきます。

フォーカスコントロール

ダイアログボックスをオープンしているとき，その中のいずれかのコントロールがハイライトされます。それがアクティブといわれる，つまり**フォーカスのある**コントロールであるという意味です。コントロールにフォーカスを当てることで，マウスを使わなくても，キーボード入力をそのコントロール用としたり，コントロールをアクティブにしたりすることができます。たとえば，あるボタンにフォーカスが合っているとき，**スペース**キーを押すことで，そのボタンを“押す”ことができます。入力行にフォーカスが合っていれば，その行に直接文字を入力することができます。

ラベルについては，本章の後のほうで説明します。

ダイアログボックス内のコントロール間は，***TAB*** キーを押すことで，フォーカスを移動させることができます。ラベルにフォーカスを合わせることはできないので，***TAB*** キーを押してもラベルはスキップされます。

TABの順番には意味があります！

TAB キーでダイアログボックス内を移動させる際には，その移動に何らかの論理的な順序を持たせたくなります。***TAB*** の順序は，オブジェクトがダイアログボックスに挿入された順番です。内部的には，ダイアログボックスにぶら下がるオブジェクトは，循環リンクリストにつながれ，最後のオブジェクトは先頭のオブジェクトにリンクするように挿入されます。

デフォルトでは，フォーカスは，最終的に一番最後に挿入されたオブジェクトのところまでやってきますが，ダイアログボックスの ***selectNext*** メンバ関数を使うか，コントロールの ***select*** メンバ関数を直接呼び出すことによって，フォーカスを別のコントロールのところにもっていくことができます。***selectNext*** を使用すれば，コントロールのリスト上を，前向きにも後ろ向きにも移動させることができます。selectNext(False) とすれば，循環リスト上を前向きに（***TAB*** の順番）進めます。selectNext(True) とすれば，後ろ向きに進めます。

好みに合わせた選択

ダイアログボックスでユーザに選択を行なわせたい場合，個々のボタンだけでは済まないことがよくあります。Turbo Vision では，ユーザがいくつかの選択肢の中から選択できるように，便利な標準コントロールをいくつか用意しています。そのうちもっとも便利なものが，**チェックボックス**と**ラジオボタン**の2つです。

チェックボックスとラジオボタンの機能は，ほとんど同じですが，チェックボタンが，いくつかあるボタンの中から複数個指定できるのに対し，ラジオボタンは，その中から1個しか選択できないという点が異なります。この2つが格好も振舞いも似ているのは，***TCluster*** という同じクラスから導出されるからです。

クラスタの作成

おそらく，直接 ***TCluster*** からインスタンスを作成する必要はでてこないでしょう。チェックボックスのクラスタの作り方とラジオボタンのクラスタの作り方は同じですので，1度だけ作り方を見ておけば充分でしょう。

ダイアログボックスを作成**してから**，ボタンを定義する**前までに**，以下のプログラムを *TMyApp::newDialog* メンバ関数に追加します。ボタンは，一番最後に挿入します。したがって，***TAB*** の順番もボタンが最後になります。

```
[ ] Hvarti
[ ] Tilset
[ ] Jarlsberg
```

```
TView *b = new TCheckBoxes( TRect(3, 3, 18, 6),
           new TSItem( "~H~varti",
           new TSItem( "~T~ilset",
           new TSItem( "~J~arlsberg", 0) )));
pd->insert(b);
```

初期化は非常に簡単です。項目を入れる矩形領域を指定し（チェックボックスそのものを表示する部分を確保することも忘れないでください），チェックボックスに続けて表示させる文字列へのポインタをリンクリストにして作成します。文字列は 0 で終端させます。

チェックボックスの値

上のプログラムでは，選択肢が 3 つのチェックボックスの組を作成しています。リストにつながる各項目に対して，まだ何の設定も行なっていません。デフォルトでは，すべてチェックされていない状態にされます。もちろん，チェックボックスのうちのいくつか，あるいは全部を初めからチェックした状態にしておきたいこともあるでしょう。リンクリストを作るときに値を設定しなくても，Turbo Vision には，見えないところで値を簡単に設定する方法がサポートされています。

一組のチェックボックスには，最大 16 個のエントリを持たせることができます。***ushort*** 型のデータメンバ ***value*** は，その 16 ビットの各ビットを各チェック項目に対応させます。

ダイアログボックス全体の構築が終わったら，次は各コントロールの値の設定と読み出しを行なうことになりますが，ここでは，コントロールを適当な場所に配置することについて話を進めます。

クラスタをもう 1 個

話を進める前に，ラジオボタンとチェックボックスとを比較できるように，ダイアログボックスにラジオボタンを 1 組追加することにします。下のプログラムは，チェックボックスの隣に 3 個からなるラジオボタンの組を作成するものです。

```
(●) Solid
( ) Runny
( ) Melted
```

```
b = new TRadioButtons( TRect( 22, 3, 34, 6),
    new TSItem( "~S~olid",
    new TSItem( "~R~unny",
    new TSItem( "~M~elted", 0 )  )));
pd->insert( b );
```

チェックボックスとラジオボタンの間の大きな違いは，ラジオボタンの場合，ひとまとまりとして扱われるボタンのうち1個しか選択できないということであり，ラジオボタンのリストの中の最初の項目がデフォルトの選択となります。

ラジオボタンの場合，すべてのボタンの状態を知っておく必要はないので（1個しかオンにできないので，**どの**ボタンがオンなのかだけを知っていればよい)，ラジオボタンのビットマップは作られません。したがって，ラジオボタンは，必要なら16個以上設けることができます。ただし，ボタンの状態は16ビットで格納されるので，1つのクラスタには，最大65,536個のラジオボタンまでということになります。この制限は大した障害にはならないでしょう。値0が，1番目のラジオボタンが選択されたことを示します。以下1が2番目のボタン，2が3番目，というぐあいになります。

コントロールのラベル付け

当然ながら，コントロールを用意しただけでは充分ではありません。選択肢の集合を表示するだけでは，ユーザは**何を**選択しているのかがわかりません。Turbo Visionでは，***TLabel***という別のコントロールを使用して簡単にコントロールに対するラベル付けを行なうメンバ関数が用意されています。

TLabelクラスには，見た目以上にいろいろな機能が備わっています。***TLabel***オブジェクトは，単にテキストを表示するだけではなく，他のビューにリンクされるようにもなっています。ラベル上をクリックすると，それにリンクされているビューにフォーカスを移します。また，ラベルにショートカット文字を定義することもできます。その文字をチルダ（~）で囲むことによって定義します。

チェックボックスにラベルを付けるには，チェックボックスをダイアログボックスに挿入した直後に，次のプログラムを書き加えます。

```
pd->insert( new TLabel( TRect( 2, 2, 10, 3), "Cheeses", b));
```

これで，“Cheeses”という語をクリックすれば，そのチェックボックスの組をアクティブにすることができます。こうすれば，よく知らない人でも，その箱の内部にある項目が cheeses に属するものであるということがわかるようになります。

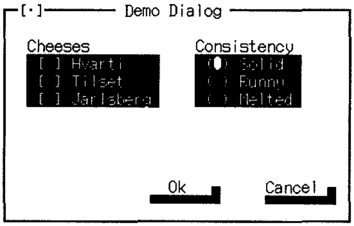
同様にして，以下のプログラムでラジオボタンにラベルを付けることができます。

これは，TVGUID14.CPP です。

```
insert( new TLabel( TRect( 21, 2, 33, 3), "Consistency", b));
```

これによって以下のダイアログボックスができあがります。

図 2.9
ダイアログボックスにラベル付きのクラスタを設けたところ



入力行クラス

以上の他に，ダイアログボックスに付けることのできる非常に簡単なコントロールがもう1つあります。文字列入力の編集を行なうための項目で，**入力行**と呼ばれるものです。入力行が行なう作業は非常に複雑なのですが，プログラマの立場からすると，***TInputLine*** は非常に使いやすいクラスです。

ラジオボタンにラベルを付けてから，ダイアログボックスを実行するまでの間に，以下のプログラムを追加します。

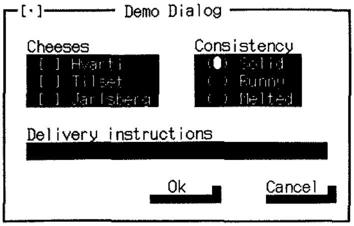
これは，TVGUID15.CPP です。

```
// 入力行を追加
b = new TInputLine( TRect( 3, 8, 37, 9 ), 128 );
pd->insert( b );
pd->insert( new TLabel( TRect( 2, 7, 24, 8 ),
                        "Delivery Instructions", b ));
```

入力行の作成は簡単そのものです。画面の範囲内で，入力行の長さを決める矩形領域を割り当てます。その他に必要な引数は，編集

する文字列の最大長だけです。この長さには，表示用の範囲を越える長さでも定義できます。***TInputLine*** クラスは，文字列を前方向にも後ろ方向にもスクロールさせることができるようになっているからです。入力行は，デフォルトで，キー入力，編集コマンド，マウスのクリックおよびドラッグを処理することができます。

図 2.10
ダイアログボックスに入力行を設けたところ



入力行には，わかりやすいようにラベルも付けられます。入力行にラベルを付けないと，ラベルなしのクラスタ以上にユーザを戸惑わせてしまうことになります。

データの設定と取得

ここまでで，かなり複雑なダイアログボックスを構築してきましたので，その**使い方**を理解しておく必要があります。つまり，これまでは，ユーザインターフェース側を設計してきたわけですが，今度は，プログラムインターフェース側を設計する必要があります。コントロールから情報を取り出す方法がわからなければ，コントロールを設けてもほとんど役に立ちません！

やらなければならないことが，基本的に 2 つあります。1 つは，ダイアログボックスをオープンしたときに初期値を設定するということであり，もう 1 つは，ダイアログボックスをクローズしたときに，設定された値を読み出すということです。ダイアログボックスが正常にクローズされないうちは，ダイアログボックス外のデータは何も修正したくはありません。つまり，ユーザがダイアログボックスをキャンセルしたら，そのダイアログボックスがオープンしていた間になされた変更は，すべて無視できなければなりません。

幸せなことに，Turbo Vision では，そのとおりのことができるようになっています。アプリケーションプログラムは，ダイアログボ

ックスがオープンされると，構造体などで表現した情報パケットをダイアログボックスに渡します。ユーザがダイアログボックスをクローズさせた場合には，アプリケーションプログラムは，ダイアログボックスがキャンセルされたのか，正常にクローズされたのかをチェックする必要があります。キャンセルされたのであれば，その構造体を修正せずに，そのまま処理を先に進めます。ダイアログボックスが正常にクローズされたのであれば，アプリケーションプログラムは，構造体をダイアログボックスに渡したのと同じ形式でダイアログボックスから読み戻すことができます。

仮想メンバ関数 ***setData*** および ***getData*** が，ビューとの間でデータをやりとりするのに使用されます。すべてのビューが，***setData*** および ***getData*** メンバ関数を備えています。

グループ（たとえば ***TDialog***）は，***setData*** 呼び出しで初期化されるとき，そのサブビューそれぞれの ***setData*** メンバ関数を呼び出してデータを渡していきます。

☞ アプリケーションからグループの ***setData*** を呼び出す場合には，そのグループ内のすべてのビューに対するデータからなる構造体を1つ渡してやります。ビューのデータは，グループにビューが挿入されたのと同じ順番に並べられなければなりません。

またデータは，各ビューごとに決まったサイズにする必要があります。すべてのビューに，そのデータ空間のサイズを返す ***dataSize*** というメンバ関数が備わっています。各ビューは，データの構造体から，***dataSize*** の量のデータをコピーし，次のビューにどこから読み始めればよいかを知らせるために，ポインタを進めます。もしサブビューのデータのサイズが間違っていた場合には，それ以降のサブビューも間違ったデータをコピーしてしまうことになります。

新たにビューを作成し，それにデータフィールドを加えていく場合には，***dataSize***，***setData***，***getData*** をオーバーライドして，ちゃんとした値を扱うようにすることを忘れないでください。データ構造内のデータの順番とサイズは，すべてアプリケーション側で管理しなければなりません。間違ったとしても，コンパイラはエラーを出力しません。

ダイアログボックスを実行したら，アプリケーションでは，ダイアログボックスがキャンセルされなかったことを確認した後で，***getData*** を呼び出して，ダイアログボックスの情報をアプリケーシ

ョンにコピーします。

以下のプログラム例では，順番に，チェックボックスのクラスタ，そのラベル，ラジオボタンのクラスタ，そのラベル，128 文字の入力行，そのラベル，およびボタン 2 個 (OK と Cancel) を初期化していきます。**表 2.1** に，これらのコントロールに必要なデータをまとめます。

表 2.1
ダイアログボックスのコントロールに必要なデータ

コントロール	データ
チェックボックス	**ushort**
ラベル	なし
ラジオボタン	**ushort**
ラベル	なし
入力行	**char**[128]
ラベル	なし
ボタン	なし
ボタン	なし

（ラベルやボタンなど）何もデータを持たないビューについては，それらのビューが ***TView*** から継承した，実際には何も行なわない ***getData*** メンバ関数を使用します。こうすることによって，このアプリケーションでは，これらのビューについては何も面倒をみなくてすみます。したがって，データを取得したり設定する場合，ラベルやボタンは，スキップすることができます。

この結果，このダイアログボックスでは，チェックボックス，ラジオボタン，および入力行の 3 つのビューだけを面倒みればよいことになります。先にも示したように，クラスタの各項目は，格納するデータとして ***ushort*** 型のデータメンバを使っています。入力行のデータは文字列です。このダイアログボックスに対する構造体は，以下のグローバル型で宣言して作成します。

```
struct DialogData
{
   ushort checkBoxData;
   ushort radioButtonData;
   char inputLineData[128];
};
```

後は，プログラムを起動したときに，この構造体を初期化し(*myApp* のコンストラクタで初期化すればよい)，ダイアログボックスに入るときにデータを設定し，ダイアログボックスが正常にクローズされたときに，そのデータを読み出せばよいわけです。先に示したように，すでに型の宣言は行ったので，次にポインタを宣言します。

```
DialogData *demoDialogData;
```

それから，ダイアログボックスの実行前後に，１行ずつ追加します。

```
// ダイアログデータをセーブして
pd->setData( demoDialogData );
ushort control = deskTop->execView( pd );
// ダイアログボックスが正しくクローズされたら読み戻す
if( control != cmCancel )
    pd->getData( demoDialogData );
```

また，このダイアログボックスに対する初期値を設定するために，*TMyApp* のコンストラクタに３行書き足します。

これは，TVGUID16.CPP の一部です。

```
demoDialogData = new DialogData;
demoDialogData->checkboxData = 1;
demoDialogData->radioButtonData = 2;
strcpy( demoDialogData->inputLineData, "Phone Mum!");
```

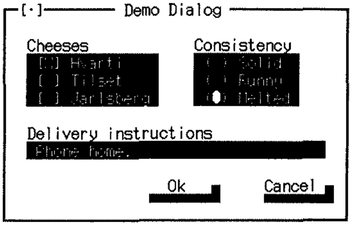


図 2.11
ダイアログボックスに初期値を設定したところ

ダイアログボックスに対してなされた変更は，すべて，ダイアログをキャンセルしないかぎり，次にそのダイアログボックスをオー

プンしたときのために，保存されるようにします。

われわれがボーランドの統合環境を開発しながら学んだことの1つに，アプリケーションは，ダイアログボックスで変更された情報を1つのオブジェクトにまとめて格納し，ダイアログボックスからのデータの設定・取得も構造体の形で行なえるようにしたほうがよい，ということがありました。こうすることによって，ダイアログボックスをオープンしようとするたびに，個々の変数からなるデータ構造体をたくさん構築しなくてもよく，またダイアログボックスから情報が返された時点で，その情報をいろいろな変数に分散させなくてもよいことになります。

ホットキーとその衝突

ラベルやチェックボックス，ラジオボタンは，ダイアログボックスの中で，別のところにフォーカスがあったとしても，デフォルトで，ホットキーに対応できるようになっています。たとえば，今回のダイアログボックスが最初にオープンされると，フォーカスはチェックボックスにあり，カーソルはチェックボックスの最初の項目にあります。ここで，“Melted”に対するショートカット ***M*** を押すと，フォーカスはすぐに Melted のラジオボタンのところに移り，それをハイライトします。

当然，ホットキーは，できるだけニーモニック(憶えやすい文字)にしたいわけですが，文字は26個，数字は10個しか使えません。したがって，衝突が起こったりします。たとえば，今回のダイアログボックスでも，***C*** を“Cheeses”，“Consistency”のホットキーにしたくなります。あるいは“Cheddar”というチーズに使うかもしれません。このような問題に対処するには，いくつかの方法があります。

まず，語の最初の文字をホットキーにしたほうがよいのですが，いつでもそうできるとはかぎりません。たとえば“Cheeses”と“Consistency”の場合，“Consistency”のホットキーを ***O*** とすることによって衝突を回避することができます。しかし，その結果，必ずしも憶えやすいものにはなっていません。もちろん，もう1つの方法として，どれかのラベルを付けかえるという手があります。“Cheeses”というラベルのかわりに，“Kind of Cheese”というラ

ベルをそのクラスタに付け，ホットキーは ***K*** にするわけです。

この手の方法では，巡りめぐってまた同じようなホットキーの衝突が起こることになります。しかしながら，衝突が，たとえばラベルとクラスタのメンバ間のものであるなら，別の手段を講じることができます。つまり，ホットキーは，ダイアログボックスの項目内に局所化することができるのです。先の例では，たとえば，各クラスタごとにホットキーを局所化すれば，チェックボックスにフォーカスがあるときには，***M*** を押しても，“Consistency” をアクティブにすることもなければ，“Melted”をアクティブにすることも**ありません**。クリックするか ***TAB*** を入力するかして，まず“Consistency” クラスタに入らなければ，***M*** はホットキーとしては機能しません。

options データメンバおよび ofPostProcess ビットについては，第4章で説明します。

すべてのホットキーは，デフォルトにすれば，ダイアログボックスで有効となります。ホットキーを局所化したいのであれば，これからダイアログボックスに挿入しようとするオブジェクトのデフォルトになっている ***options*** データメンバを書き換えればよいのです。たとえば，チェックボックス内のホットキーを局所化したいのであれば，ダイアログボックスに挿入する前に，以下のように1行書き足します。

```
TView *b = new TCheckBoxes( TRect( 3, 3, 18, 6),
    new TSItem( "~H~varti",
    new TSItem( "~T~ilset",
    new TSItem( "~J~arlsberg", 0 )
    )));
(b->options) &= (!ofPostProcess);  // オフにする
pd->insert( b );
```

これで，ホットキー ***H***，***T***，および ***J*** は，クリックか ***TAB*** によって “Cheeses” クラスタに入った後でなければ機能しません。それでも，***GRPH-H***，***GRPH-T***，および ***GRPH-J*** は，それまでどおり機能しつづけます。

詳細については，第5章の180ページを参照してください。

ラベルには，絶対にフォーカスはこないということに注意してください。したがって，ラベルに対してホットキーを働かすためには，その ***ofPostProcess*** ビットをオンにする必要があります。

ofPostProcess をセットすることにより，ユーザは，ダイアログへの情報の入力を速く行なうことができることになります。しかし，これが欠点となることもあります。ユーザが，あるところに行こうと思ってホットキーを押したら，衝突があったために，どこか他の

ところに行ってしまうという場合です。同様に，ユーザがホットキーがアクティブだと**思って**いても，それが局所的にしかアクティブでなかった場合，その有効な範囲外でホットキーが押されても，何も行なってくれず，混乱を招くかもしれません。

ダイアログボックスに衝突がないか，注意深くテストしてみることをお勧めします。できることなら，ホットキーを重複させないようにし，またユーザがいつでも，どの選択肢が有効なのかがわかりやすいようにすべきです。

その他のダイアログボックスコントロール

今回の例ではとりあげませんでしたが，ダイアログボックスには，その他にも，いくつか出来合いの部品が用意されています。それらは，これまで使ってきた項目と同じようにして使用することができます。インスタンスを新たに作成し，それをダイアログボックスに挿入し，そしてデータ構造体に必要なデータをすべて持たせてやればよいわけです。この節では，それぞれの機能と使用法を簡単に説明しておきます。もっと詳細なことが，第13章**『クラスリファレンス』**に示されています。

固定文字列

TStaticText は，渡されてきた文字列を単に表示するだけのビューです。文字列は，ビューの矩形領域内でワードラップします。文字列が '¥x03' で始まる場合には，テキストはセンター揃えになり，'¥n'で改行できます。デフォルトにすると，テキストにフォーカスをもってくることはできません。また，このオブジェクトに対しては，データ構造体からのデータはありません。

リストビューアー

TListViewer は，1段もしくは何段かの縦方向のリストを表示するもので，その中からユーザが項目を選択できるようになっています。個々の ***TListViewer*** は，2つのスクロールバーと通信することもできます。

TListViewer は，ビルディングブロックとして作られたもので，それだけでは利用できません。リストを扱う機能は持っていますが，それ自身はリストを持ちません。その抽象メンバ関数 ***getText*** が，その ***draw*** メンバ関数用にリストのメンバをロードします。***TList-Viewer*** から導出したクラスを作って機能させるためには，そのオ

ブジェクトで実際に使われるデータをロードしてこられるように，***getText*** をオーバーライドします。

リストボックス

TListBox は，***TListViewer*** から導出したクラスで，実際に機能するオブジェクトです。***TCollection*** を備え，***TCollection*** には，文字列へのポインタを入れることになっています。***TListBox*** は，スクロールバーは１個しかサポートしていません。リストボックスの実例は，ボーランドの統合環境のファイル選択リスト，あるいはディスクで提供される FILEVIEW.CPP のサンプルプログラムに見ることができます。

リストボックスのデータの取得および設定は，***TListBoxRec*** 型構造体を利用することで，非常に簡単に行なうことができるようになっています。***TListBoxRec*** 型には，表示される文字列のリストを集めたものに対するポインタ，およびリストの中のどの項目が現在選択されているかを示すワードが含まれています。

ヒストリー

THistory は，入力行および関連するリストボックスとともに機能するオブジェクトです。入力行の横にある矢印アイコンをクリックすることによって，これまで入力行に入力されてきたデータのリストを表示させ，そのリストの中からデータをどれでも選択できるようになっています。これによって，同じデータを再度打ち込む手間を少なくしたり，省くことさえも可能になります。

THistory クラスは，ボーランドの統合環境においても，File | Open ダイアログボックスや Search | Find ダイアログボックスなど，いろいろなところで利用されています。

標準ダイアログボックス

STDDLG モジュールには，*TFileDialog* と呼ばれる出来合いのダイアログオブジェクトが含まれています。このダイアログボックスは，統合環境でファイルをオープンする際に利用されているものです。*TFileDialog* は，さらに，STDDLG 内の他のクラスをいろいろと使っています。これらのクラスも重宝することができると思います。

```
class TFileInputLine: public TInputLine
class TFileCollection: public TSortedCollection
class TSortedListBox: public TListBox
class TFileList: public TSortedListBox
class TFileInfoPane: public TView
```

第 15 章『**標準ダイアログボックスの実現**』が，この一連のクラスについてのリファレンスとなっています。

第2部

Turbo Visionのプログラミング

第3章
クラス階層

この章は，このマニュアルの第1部を読み終えて，Turbo Visionの基本思想，機能，用語法について，おおすじを理解していることを前提としています。またTurbo　Visionを利用するためには，C＋＋を実際に開発して使用した経験も必要です。

この章では，OOPおよび階層について，一般的なことをいくらか説明した後，さっそくTurbo Visionのクラス階層に入り，クラス同士が継承のメカニズムを通してどのように関連付けられるかについて重点的に説明します。標準的なクラスが備えるおもな性質（クラスの多くには，それとわかるようなクラス名がつけられている）を理解すれば，各クラスが，継承したり新たに追加したメンバをいろいろと組み合わせながら，機能を発揮していることがわかるようになります。

階層の概要

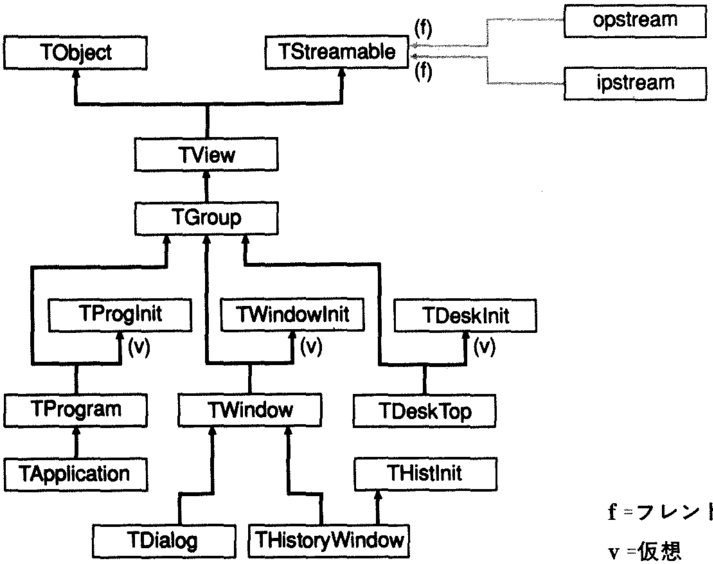
第13章『**クラスリファレンス**』では，標準クラスごとに，メンバ関数とデータメンバについて詳細に説明していますが，クラスの階層がどのような構造となっているかを全体的に把握しておかないと，細かい説明がいっぱいありすぎて，すぐにうんざりしてしまうと思います。そこでこの章では，Turbo Vision の階層全体を，形式ばらずにざっと眺めてみることにします。第2部の残りの部分では，Turbo Vision の各要素とその使用法についてもっと詳細に説明します。第3部は，クラスとそのメンバのリファレンスです。

TObject は，Turbo Vision のほとんどのクラスのルートクラスとなっています。また一方，多くのクラスが ***TView***（***TObject*** の直接の導出クラス）から直接導出されています。

TObject/***TView*** の階層の概要を**図 3.1**，**図 3.2** に示します。これらの図は，後で役に立ってくるのでしっかりと観察しておいてください。たとえば，***TDialog*** は ***TWindow*** から導出され，一方 ***TWindow*** は ***TGroup*** から，また ***TGroup*** は ***TView*** から導出されていることがわかれば学習曲線はうんと高まります。導出された新しいクラスは，その利用方法をすでによく理解できている性質を継承しています。したがって，そのクラスについては，親から継承したもの以外に新たに付加されたデータメンバや性質についてだけ学べばよいことになります。

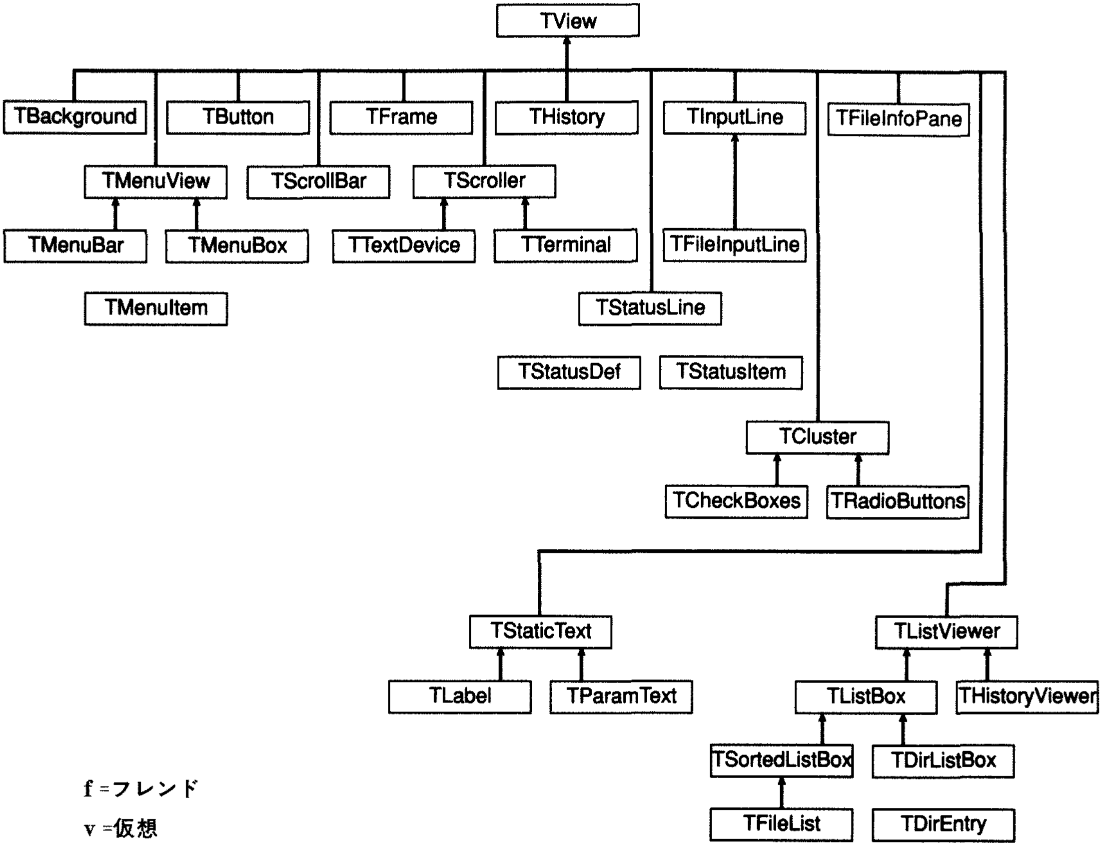
階層には，多重継承の例がいくつかあります。たとえば ***TProgram*** が，***TGroup*** と ***TProgInit*** の両方から導出されています。C++ の慣例にしたがって，矢印は基本クラスに**向かって**走っています。これらの図の中で他と何も接続のないクラスは，その階層と近い関係にはあるが，その階層の一部ではないクラスです。それらのクラスの近くにあるクラスと近い関係にあるクラスです。

図 3.1
クラス階層の概要



f =フレンド
v =仮想

図 3.2
Tiew のクラス階層



f =フレンド
v =仮想

階層の作成について

どういうアプリケーションであっても，“完璧な”階層というものはありません。すべてのクラス階層が，いろいろと実験を重ねてみて得られる妥協の産物です（経験を重ねていくうちに，感覚がいろいろと身についてきます)。われわれがクラス型の階層を開発する上で経験したことを利用して，みなさんは独自のクラス階層を開発することができます。もちろん，Turbo Vision に用意してある標準的なクラス以外にも，独自の基本クラスを作成し，何か特別な働きをさせることもできます。

詳細に関しては後で説明しますが，OOP プロジェクトの場合ではいつもそうなのですが，まず新しく作成するクラスについて全体的に設計しておくことが成功の秘訣です。

クラスの分類

Turbo Vision では，すべてのクラスが対等な関係にあるわけではありません。クラスの機能は，基本クラス，ビュークラス，ミュートクラスの3つのグループに分けることができます。この章では，これらのグループごとに節を分けて説明します。

各グループの中には，また，いろいろな種類のクラスがあり，インスタンス化（つまり，そのクラスのオブジェクトを生成）して使うクラスと，関連する使いやすいクラスを導出するための基礎としての役割しかもたない“シード”クラスとがあります。Turbo Vision の階層内にあるクラスを見ていく前に，この“シード”クラスというものについて少し説明しておきましょう。

シードクラスのことを，OOP の世界では“抽象クラス”ということもあります。C++ における抽象クラスとは，厳密には，純粋仮想関数を少なくとも1個用意しているクラスのことです。C++ の抽象クラスは，インスタンス化することは**できず**，単に抽象クラスまたは非抽象クラスを導出するためのベースとして存在します。Turbo Vision には，インスタンス化して実際に使用されることはないが，C++ の技術的な意味からすると抽象クラスではないシードクラスがいくつかあります。

シードクラス

クラスの中には，シードとして使われるものが数多くあります。それをベースにして，特殊なクラスを導出して，すぐに利用できるようにするものです。シード型を設ける理由は，概念的にそうしたほうがよいということもありますが，コーディングの手間を減らすという実際的な目的が大半を占めます。

TRadioButtons 型と ***TCheckBoxes*** 型を例にとってみましょう。いずれも，***TView*** から導出しようと思えば，簡単に導出することができるでしょう。しかし，この両者には，共有するものがたくさんあります。この共通性が，***TCluster*** というシードクラスを必要

としているのです。つまり，***TRadioButtons*** と ***TCheckBoxes*** は，同じ ***TCluster*** から導出され，特殊なメンバ関数をいくつか付加することによって，それぞれに固有な機能を持たせてあるのです。

シードクラスは，そのままインスタンス化しても役には立ちません。たとえば，***TCluster*** や *MyCluster* の場合，役に立つ ***draw*** メンバ関数は備えていません。***TView::draw*** を継承しているだけで，オーバーライドしていないので，*MyCluster::draw* は，デフォルトの色を使って，単なる矩形領域を表示するだけです。好みに合わせてコントロールのクラスタを設け，それにラジオボタンやチェックボックスとは異なる性質を持たせたいのであれば，***TCluster*** から *TMyCluster* を導出すればよいでしょう。あるいは，***TRadioButtons*** か ***TCheckBoxes*** のうち，自分の要求に近いほうから専用のクラスタを導出したほうが簡単かもしれません。いずれにせよ，データメンバを追加し，メンバ関数も追加するかオーバーライドすることになりますが，あまり手間をかけずに済みます。クラスタのファミリーをまるごと好みに合わせたいというのであれば，中間にもう一段シードクラスを設ければ都合がよいでしょう。

空のメンバ関数

あるクラスをインスタンス化して使いものになるかどうかは，その場の状況によります。Turbo Vision の標準的なクラスのうち多くのものが空のメンバ関数を備えており，その関数については導出クラスで定義する**必要があります**。また標準的なクラスには，最小限のデフォルト動作を行なうスケルトンメンバ関数というものを備えているものもあります。これが必要な機能を果してくれる場合もあります。それがない場合には，型を導出する必要があります。

一般には，Turbo Vision の階層を下っていくにつれ，標準的なクラスは次第に特殊化されていきます。クラスの名前は，データメンバやメンバ関数の中にカプセル化された機能を表わしています。アプリケーションに“標準的な”インターフェースを持たせる場合，たとえば，デスクトップ，メニューバー，ステータスライン，ダイアログボックスをいくつか持たせる場合，たいてい，それらしい名前を持つ基本クラスを利用すれば用が足ります。

クラスのインスタンス化と導出

クラスに対しては，基本的に2つの使い方があります。クラスのインスタンスを作成する（それをインスタンス化する）場合と，それに対する新しいクラスを作成する場合の2つです。後者の場合，クラスが得られたことになり，そのクラスに対して，また，上の2つの使い方を適用することができます。これらの使い方について，もう少し詳しく見てみることにしましょう。

インスタンス化

クラス（オブジェクトともいう）のインスタンスを作成するには，通常，静的なもしくは動的な変数を宣言します。

```
myScrollBar(TRect r);
TButton *someButton = new TButton( ... );
```

MyScrollBar は，***TScrollBar*** のコンストラクタ によって初期化され，デフォルトのデータメンバ値が与えられます。このことは，第13章『**クラスリファレンス**』に示される ***TScrollBar*** のコンストラクタのところを見れば，確かめることができます。***TScrollBar*** は，***TView*** の導出クラスなので，***TScrollBar*** のコンストラクタは，***TView*** のコンストラクタを呼び出し，***TView*** から継承したデータメンバの設定を行なわせます。同様にして，***TView*** のコンストラクタは，***TObject*** の導出クラスですので，***TObject*** のコンストラクタを呼び出し，メモリ割り当てを行なわせます。***TObject*** には親がないので，遡っていくのはそこまでです。

オブジェクト *myScrollBar* は，これで，デフォルトのデータメンバ値が設定されたわけで，必要に応じてこの値を変更することになります。また，***TScrollBar*** のメンバ関数，および（オーバーライドされているかもしれませんが）***TView*** と ***TObject*** のメンバ関数全部を継承しています。*myScrollBar* を使うためには，それが備えているメンバ関数，とくに ***handleEvent*** と ***draw*** が何をする

のかを知っておく必要があります。必要な機能が，***TScrollBar*** で定義されていなければ，新しい型を導出する必要があります。

導出

次のようにして既存のクラスから新しいクラスを簡単に導出することができます。

```
class TNewScrollBar: public TScrollBar
{
// 新しいメンバをここで宣言
};

// 新しいメンバをここで定義
```

まだこの新しいクラスのすべてのインスタンスを備える必要はありません。*TNewScrollBar* のクラスを作成する前に，新しいメンバ関数を定義したり，あるいは ***TScrollBar*** のメンバ関数を必要に応じてオーバーライドしなければなりません。また，新たにデータメンバをいくつか追加する必要もあるかもしれません。データメンバを追加しないのであれば，スクロールバーのクラスを新たに導出する必要はないでしょうから。メンバを新たに追加したり，メンバの機能を変更することが，***TScrollBar*** の機能を拡張していく過程なのです。新たに付加した コンストラクタが，新しいスクロールバーのクラスに対するデフォルト値を決めることになります。

メンバ関数

Turbo Vision のクラスメンバ関数は，4種類に分けることができます(ある程度ダブる部分もありますが)。それぞれについて，以下で説明します。

空のメンバ関数

空のメンバ関数は，基本として使用されるクラスで定義されるもので，インスタンスは定義されていません（あるいは，不正呼び出しをトラップするための文が書かれている)。空のメンバ関数は，常に仮想で導出クラスで機能を定義してから使用する**必要があります。**

デフォルトメンバ関数

デフォルトメンバ関数も，やはり基本クラスで定義されるものであり，最小限の動作だけが定義されています。たいていは，導出クラスでオーバーライドされますが，階層チェイン中のすべてのクラスに対し，デフォルトとして，そこそこの働きをします。たとえば ***TSortedCollection::compare*** がこれにあたります。

仮想メンバ関数

仮想メンバ関数は，そのクラス宣言において virtual の指定をしたものです。仮想メンバ関数は，導出クラスでオーバーライドすることができます。オーバーライドする関数は，元のオーバーライドされる関数の引数並びと戻り値の型を一致させる必要があります。オーバーライドしてできた関数自身は，virtual のキーワードを指定するしないに関わらず仮想関数となります。仮想関数は，オーバーライドする必要はありませんが，通常は，どこかでオーバーライ

ドして使うことを目的としています。たとえば ***TView::dataSize*** および ***TGroup::handleEvent*** がこれにあたります。C++ の仮想関数は，**多態性**の概念を実現するものです。関数および関数が呼び出したメンバ関数は動的に，すなわち実行時にオブジェクトと結合させることができます。

仮想でないメンバ関数

仮想でないメンバ関数は，通常，オーバーライドされません。導出クラスでは，必要なら，同じ名前で引数と戻り値の型の異なるメンバ関数を定義することによって，メンバ関数をオーバーロードしたり，隠蔽することができます。ただし，仮想でないメンバ関数は，多態的な働きはしません。このことは，メンバ関数を呼び出す場合に，非常に重要な意味を持ってきます。たとえば，*pGeneric* が ***TView**** 型のポインタ変数だったとすると，***TClassName*** が ***TView*** の導出クラスである場合には，*pGeneric* に ***TClassName*** *型のポインタを代入することもできます。しかしながら，そのポインタを逆参照して仮想でないメンバ関数を呼び出した場合，常に ***TView*** のメンバ関数が呼び出されることになります。なぜならコンパイル時にそのポインタの型が ***TView*** に決定されているからです。つまり，*pGeneric->non_virtualMemberFunction* は，**常に** *TView::non_virtualMemberFunction* と等しいのです。たとえ何か別の型のポインタを *pGeneric* に代入したとしてもです。これに対して仮想関数の場合は，*pGeneric->virtual_MemberFunction* は，実行時のポインタの型に注意して，**pGeneric* オブジェクトで定義されている関数を呼び出します。その場合，***TView*** から導出された型の関数を呼び出すこともできます。

静的メンバ

静的メンバは(データの場合も関数の場合も)，クラス本体の宣言で ***static*** キーワードを使用して宣言します。静的メンバ変数については，そのクラスのすべてのオブジェクトが1個の実体を共有し

ます。静的メンバ関数は，this ポインタを持たず仮想関数にすることはできません。静的メンバは，Turbo Vision において重要な働きをします。たとえば後で出てきますが，***TProgram::initStatusLine*** は，静的データメンバである ***TProgram::statusLine*** を作成する静的メンバ関数となっています。

Turbo Vision のデータメンバ

TView，***TGroup***，***TWindow*** の三大クラスを使用する場合，それらの中にどのようなデータメンバがあるのかを見ておくことで，階層の実際の働きがわかるようになります。また，階層を下っていくにつれて機能が拡張されていく様子が少しながらわかってきます（クラスツリーは，根から下位方向に成長します！）。

表 3.1
ビューがデータメンバを継承していく様子

TView データメンバ	*TGroup* データメンバ	*TWindow* データメンバ
cursor	*cursor*	*cursor*
dragMode	*dragMode*	*dragMode*
eventMask	*eventMask*	*eventMask*
growMode	*growMode*	*growMode*
helpCtx	*helpCtx*	*helpCtx*
next	*next*	*next*
options	*options*	*options*
origin	*origin*	*origin*
owner	*owner*	*owner*
size	*size*	*size*
state	*state*	*state*
	buffer	*buffer*
	clip	*clip*
	current	*current*
	endState	*endState*
	last	*last*
	lockFlag	*lockFlag*
	phase	*phase*
		flags
		frame
		number
		palette
		title
		zoomRect

TGroup は，***TView*** のデータメンバをすべて継承した上で，グループを機能させるために必要なデータメンバをいくつか追加しています。たとえば，グループ内で現在処理中のビューおよびその前に処理していたビューに対するポインタなどです。一方，***TWindow*** は，***TGroup*** のデータメンバをすべて継承した上で，さらに，ウィンドウの処理に必要なデータメンバをいくつか追加しています（たとえば，タイトル，ウィンドウフレームへのポインタ，ウィンドウの数などです）。

TWindow をよく理解するためには，ウィンドウが１つのグループであり，１つのビューであることを頭に入れておいてください。

基本的なクラス

Turbo Vision には，基本的なクラスがいくつかあり，他のクラスから使用されたり，より複雑なクラスの階層のベースとして使われたりします。***TPoint*** と ***TRect*** は，Turbo Vision の階層中のすべての可視クラスが使用します。***TStatusItem*** と ***TMenuItem*** は，ステータスラインとメニューを作成するために使用されます。また，***TObject*** は，ビュー階層の基本となるものです。

これらの型を持つクラスは，直接は表示されません。***TPoint*** は，単に画面の位置を示すクラス（*x, y* 座標）です。***TRect*** は，ビュークラスに似ていますが，矩形領域の境界として左上隅と右下隅の座標を持っているだけで，表示を伴わないユーティリティメンバ関数とオーバーロード演算子をいくつか備えています。

TPoint

このクラスは，点を表わします。そのデータメンバ *x*，*y* で，画面上の位置を示す直交座標 *(x, y)* を定義します。点（0，0）は，画面の左上隅を表わします。*x* は右方向，*y* は下方向の座標です。***TPoint*** では，+，==，+= などのオーバーロード演算子をいくつか定義しており，自然な形で演算を行なうことができるようになっています。他のクラスでは，グローバル座標（画面全体）をローカル座標（ビューの原点からの相対座標）に変換するメンバ関数が用意されています。

TRect

このクラスは，矩形を表わします。そのデータメンバ *a*，*b* は，矩形の左上隅（ *a.x, a.y* ）と右下隅（ *b.x, b.y* ）の点を定義する ***TPoint*** クラスです。***TRect*** は，メンバ関数と演算子として，***move***，***grow***，***intersect***，***union***，***contains***，　==，!=，および

is Empty を備えています。***TRect*** コントラクタは，整数を 4 個または，***TPoint*** オブジェクトを 2 個引数にとります。***TRect*** オブジェクトは，表示されるビューではなく，それ自身で表示することはできません。ただし，ビューはすべて矩形なので，各ビューのコンストラクタは，***TRect*** 型の引数である *bounds* を用いて，ビューがサポートする領域を決定します。

TObject

TObject は，Turbo Vision のほとんどのクラスの基本クラスとなっています。***TObject*** は，デストラクタの他に，***destroy*** と ***shutDown*** の 2 つのメンバ関数を用意しています。***destroy*** は，C＋＋ 標準の演算子 delete のかわりをする関数です。***shutDown*** は，***destroy*** が，導出したクラスや関連するクラスを適切に消去するために内部的に使用します。

TObject の導出クラスは，ビューかビューでないかのどちらかになります（以下ビューでないクラスを「非ビュー」といいます)。ビューは，***TView*** の導出クラスです。***TView*** は，ビューに非ビューにはない特有の性質を与えます。たとえば，ビューは，自分自身を表示したり，ビューに送られてきたイベントを処理することができます。非ビュークラスは，ストリーム処理用ユーティリティや，ビューなどの別のクラスに対してホストの役割をするもので，画面で直接“見ることはできません”。

ビュー

TObject の導出クラスで，表示可能なものがビューです。つまり ***TObject*** の直接の導出クラスである ***TView*** から導出されるクラスです。ここで“見える (visible)”ということと“表示可能である (displayable)”ということは，区別されます。ビューが，別のビューによって，全体なり，一部なりが見えなくなる場合もあるからです。

ビューの概要

ビューとは，画面上の矩形領域に描かれる（表示される）クラスのことです。ビュークラスは，***TView*** の導出クラスでなければなりません。***TView*** 自身は，画面上の空の矩形領域を表わすシードクラスです。***TView*** を基本クラスとすることにより，導出されたビューには，画面上に少なくとも1個の矩形領域と最小限の機能を備えた仮想メンバ関数 ***draw*** が持たされることになります（直系導出クラスは，すべて，固有の ***draw*** メンバ関数を用意しなければならない)。

Turbo Vision を使ってプログラムを開発する場合には，たいてい，***TView*** よりももっと専用化した導出クラスを使うことになるでしょうが，***TView*** が備える機能は，Turbo Vision 全体で効力を持ちますので，***TView*** が何を行なうのかについて，理解しておく必要があります。

グループ

TView の重要性は，99, 100ページにある**図 3.1** に示した階層図からも明白です。Turbo Vision のアプリケーションを実行してみて，そこに見ることのできるものは，すべて何らかの形で ***TView*** から導出されたものです。しかし，画面上に見ることのできるこれ

らのクラスの中には，別の理由で重要なものがあります。それは，いくつかのクラスが協調できるような仕掛を持ったクラスです。

TGroup クラス

TGroup は，相互作用を行なう関連のあるサブビューを，リストとして動的にチェインするためのものです。この場合，あるビューがそのグループの**オーナー**となり，それを介して処理が行なわれます。各ビューには，****TGroup*** 型の ***owner*** データメンバがあり，そのビューのオーナーである ***TGroup*** クラスをポイントします。ポインタがヌルの場合には，そのビューにはオーナーがないことを表わします。***next*** というデータメンバで，チェイン内の次のビューにリンクします。グループはビューですので，サブビュー自身がグループとなり，さらにサブビューを抱えるというようなこともあります。

チェインの状態は，アプリケーションを実行している間は，ユーザがクリックしたりキー入力を行なうことにより，絶えず変化します。グループを新たに作成したり，サブビューをグループに追加(挿入）したり，グループから削除したりすることもできます。グループが機能している間は，あるビューに対して何か操作を行なうことによって，別のビューが見えなくなったり現われてきたりするようにすることができます。したがって，グループには，いろいろな動作を協調させる機能が必要です。

デスクトップ

TDesktop は，正常に起動したときに背景となるビューであり，ユーザのデスクトップとしておなじみのものです。通常は，メニューバーとステータスラインが上下に付きます。一般に，***TApplication*** は，***TDesktop***, ***TMenuBar*** および ***TStatusLine*** クラスを含むグループのオーナーとなります。他のビュー（ウィンドウやダイアログボックス）は，ユーザの操作（マウスやキーボードからのイベント）に応じて，デスクトップ上に構築され，表示され，操作の対象となります。アプリケーションの実際の仕事は，ほとんどがデスクトップ上で行なわれることになります。

プログラム

TProgram は，その導出クラスである ***TApplication*** に対して，いろいろな仮想メンバ関数を用意します。***TProgram*** は，***TGroup*** と ***TProgInit*** の2つを基本クラスとしています（多重継承)。

アプリケーション

TApplication は，Turbo Vision のアプリケーションを作成する際に，プログラムのテンプレートとなるクラスです。これは，***TGroup*** の導出クラスです（***TProgram*** を介して)。通常，***TApplication*** は，サブビューとして，***TMenuBar***，***TDeskTop***，および ***TStatusLine*** を持つことになります。したがって，これら3つのサブビューを作成したり挿入するためのメンバ関数を備えています。***TApplication*** で重要な役割を果たすメンバ関数は，***TApplication::run*** です。これが，アプリケーションプログラムを実行します。

ウィンドウ

TWindow オブジェクトは，それにリンクされる ***TFrame*** オブジェクトの力を借りて，枠つきの矩形領域として表示されるおなじみのウィンドウです。***TView*** から継承したメンバ関数を使用して，ドラッグしたり，リサイズしたり，画面から消したりすることができます。***frame*** というデータメンバが，そのウィンドウの ***TFrame*** クラスをポイントします。また ***TWindow*** クラスは，自前のメンバ関数を使って，それ自身をズームしたり，クローズしたりすることができます。***TWindow*** では，ウィンドウ内のサブビューを選択するときに，次のサブビューや前のサブビューを選択できるように，***Tab/Shift-Tab*** キーの処理を行なうことができます。***TWindow*** のイベントハンドラは，クローズ，ズーム，リサイズの各コマンドの処理を行ないます。番号つきのウィンドウは，ホットキー ***GRPH-n*** で選択できます。

ダイアログボックス

TDialog は，***Twindow*** の導出クラスであり，ダイアログボックスを作成し，ユーザとの間のいろいろな対話を処理するために使用されます。通常，ダイアログボックスには，ボタンとかチェックボックスなどのコントロールが付けられます。親のメンバ関数 ***execView*** を使って，直前の（***previous***）コンテキストをセーブし，グループに ***TDialog*** クラスを挿入した後，ダイアログボックスをモードつきにします。これによって，***TDialog*** クラスは，ボタンのクリックだとかキー入力などの，ユーザの操作によって発生したイベントを処理するようになります。とくに，**ESC** キーは キャンセル（***cmCancel***）で，**リターン** キーは ***cmDefault*** ブロードキャストイベント（通常は，デフォルトボタンが選択されたという意味）で処理します。終了時には，***execView*** が先にセーブされたコンテキストを元にもどします。

ターミナルビュー

ターミナルビューとは，グループ以外のすべてのビューのことです。つまり，他のビューのオーナーとなることのできないビューです。したがって，ビューのチェインの終端となります。

フレーム

TFrame は，***TWindow*** クラスの枠（境界）として表示されるもので，ウィンドウを移動したりクローズしたりするためのアイコンもいっしょに付けられます。***TFrame*** クラスは，決して単独で使用されることはなく，常に ***TWindow*** と一緒に使用されます。

ボタン

TButton クラスは，タイトルつきのボックスで，“押された”ときに特定のコマンドを発生します。通常は，ダイアログボックス内

に置かれ（ダイアログボックスがオーナーとなり），“OK”とか“Cancel”などの選択用に使用されます。ダイアログボックスは，通常，最初に表示された時点では，モードつきビューであり，したがって，ボタンイベントなど，すべてのイベントに対してトラップをかけ，処理を行ないます。***TButton*** のイベントハンドラは，ボタンを押す方法として，ボタンの矩形領域のクリック，ホットキーの入力，もしくは**リターン**キーによるデフォルトボタンの選択を用意しています。

クラスタ

TCluster は，チェックボックスおよびラジオボタンを作成するためのシードクラスです。クラスタとは，グループのなかのすべてのコントロールが同じ方法で処理される場合のグループのことです。クラスタを構成するコントロールは，***TLabel*** クラスと組み合わせて使うことが多く，その場合，コントロールの隣に内容を示すために付けられたラベルを選択することによって，コントロールを選択することができます。その他，データメンバには，ユーザ定義の値を入れておくための ***value***，およびそのクラスタ内のコントロールのうち選択されたものを示すための ***sel*** があります。メンバ関数としては，文字で構成されるアイコンや“マーク”文字を表示するためのものが用意されています。クラスタ内のコントロールをマークするには，カーソルキーを使うか，マウスクリックによります。

ラジオボタンは，そのクラスタ内の１つのコントロールしか選択することのできない特殊なクラスタです。他のラジオボタンを選択すると，それまで選択されていたラジオボタンは無効になります（ラジオの選局ボタンと同じです）。チェックボックスは，そのクラスタ内のコントロールをいくつでもマークする（選択する）ことのできるクラスタです。

メニュー

TMenuView と，その導出クラスである ***TMenuBar*** および ***TMenuBox*** は，プルダウンメニューや何段階にでもネストさせることのできるサブメニューを作るための基本的なクラスです。アプ

リケーション側からは，メニュー選択用の文字列（ホットキーをハイライトさせることもできる）と，それぞれの選択に対して発生させたいコマンドとを与えてやります。***handleEvent*** メンバ関数が，マウスや (and/or) キーボード（ショートカットやホットキーなど）によるメニュー選択の処理を行ないます。

メニュー選択は，通常，***TApplication*** クラスが所有する ***TMenuBar*** クラスを使用して，表示されます。メニュー選択は，***TMenuBox*** 型のクラスの中に表示されます。

たいていのアプリケーションでは，直接メニュークラスに関わることはありません。***TApplication::initMenuBar*** をオーバーライドし，***newTMenuBar, newTSubMenu, newTMenuItem*** および ***newLine*** 呼び出しをネストさせることにより目的に合わせた初期化を行なえば，***Turbo Vision*** は，それに応じてメニューを作成し，表示し，対話処理を行なってくれます。

ヒストリー

シードクラス ***THistory*** は，汎用的なピックリストの機能を実現するためのものです。追加されたデータメンバは ***link*** および ***historyId*** の２つです。***link*** は ***THistory*** クラスにリンクされた ***TInputLine*** を表わし，***historyId*** は，その入力行でそれまでに入力されたデータのリストに対する ID を表わします。***THistory*** は，***THistoryWindow*** および ***THistoryViewer*** と組み合わせて使用されます。

入力行

TInputLine は，入力行の形で基本的な文字列エディタの機能を果たす特殊なビューです。通常のキー入力やカーソル移動（***HOME***（行頭へ移動）や ***HELP***（行末へ移動）を含めて）なら，すべて処理することができます。削除や挿入の機能のほか，挿入モードか上書きモードかを選択したり，自動的にカーソルの形を変える機能も備えています。マウスをドラッグして，テキストのブロックをハイライトさせることもできます。

リストビュアー

TListViewer クラスもシードクラスです。このクラスを基にして，***TListBox*** など，いろいろな種類のリストビュアーを導出することができます。***TListViewer*** のメンバを使用して，文字列のリンクリストを表示することができます。また，スクロールバーを1個もしくは2個使用して，表示の制御を行なうことができます。イベントハンドラは，リスト上の項目をマウスまたはキーボードで選択できるようにします（選択されると項目がハイライトする)。***draw*** メンバ関数は，リサイズやスクロールの処理を行ないます。***TListViewer*** には，空の仮想メンバ関数 ***getText*** が用意されており，項目として表示されるテキストを作成したり，編集したりするための実体を作り込んでやる必要があります。

TListViewer から導出される ***TListBox*** は，ごく一般的に使用されるリストボックスを作成するもので，ファイル名などの文字列をリストにして表示します。***TListBox*** クラスは，そのような項目のリストを，一段あるいは何段かにして表示します。縦方向のスクロールバーを持たせることもできます。ただし，***TListViewer*** にある横方向のスクロールバーはサポートされません。***TListViewer*** から継承したメンバ関数を使用して，マウスもしくはカーソルキーによる項目の選択（およびハイライト）をサポートします。***TListBox*** では，***list*** というメンバ関数が追加されており，***TCollection*** クラスをポイントするために使用されます。***TCollection*** は，リストにして表示したり選択したりするための項目となります。どのような内容の項目を用意するかは，アプリケーション側で決定します。アプリケーション側の責任という意味では，項目が選択されたときに，どのような処理を行なうかについても同様です。

スクロールクラス

TScroller クラスとは，スクロール可能なビューのことであり，“背景にある”別のもっと広いビューの上で，窓口としての役割を果たすビューです。スクロールには，キーボード入力に応じて，あるいは一緒に組み合わせて使用する ***TScrollBar*** オブジェクトに

対する操作に応じて行なわれます。スクロールには，それが管理する横方向および縦方向のスクロールバーを示すための，*hScrollBar* および *vScrollBar* の2つのデータメンバがあります。また ***TScroller*** の ***delta*** データメンバは，リンクしているスクロールバーにあるデータメンバとの関係から，*x* 方向，*y* 方向の単位スクロール幅を決めます。

TScrollBar クラスは，縦方向にも横方向にも使用できるコントロールです。おもなデータメンバとしては，***value***(スクロールバーインジケータの位置)，***pgStep***（マウスクリックおよび **ROLLUP**, **ROLLDOWN** キーの入力に応じてスクロールさせる幅），***arStep***（マウスクリックおよびカーソルキー入力に応じてスクロールさせる幅）があります。

スクロールクラスおよびそれに付加されるスクロールバーは，通常，***TWindow*** クラスの管理下に置かれ，複雑に関係してくるいろいろなイベントも ***TWindow*** によって処理されます。たとえば，ウィンドウをリサイズした場合には，それに応じて，スクロールクラスも再表示させなくてはなりません。また，スクロールバーの各値を更新し，再表示する必要があります。

テキストデバイス

TTextDevice は，スクロール可能な ***TTY*** 型のテキストビューアー/デバイスドライバです。***TScroller*** から継承したデータメンバやメンバ関数とは別に，***TTextDevice*** では，デバイスに文字列を書き込むための仮想関数が定義されています。***TTextDevice*** は，実際のターミナルドライバを導出するためのベースとなる型としてあるものです。***TTextDevice*** は，***TScroller*** のコンストラクタとデストラクタを使用します。

TTerminal は，文字列をバッファ化して書き込む“ダム”ターミナルです。バッファのサイズは，初期化時に決定されます。

固定文字列

TStaticText クラスは，単に，***text*** データメンバで指定する固定の文字列を表示するだけのビューです。イベントを受けても，何

もしません。***TLabel*** クラスとして宣言することによって，ラベルとして使用されるテキストを保持するこのビューを，マウスクリックやカーソルキー，**GRPH-**文字キーの入力で選択する(ハイライトさせる）ことができるようになります。追加されたデータメンバ *link* は，ラベルを他のビュー（通常は，ラベル関係のすべてのイベントを処理するコントロールビュー）にリンクするために使用されます。ラベルを選択することによって，リンクされたコントロールが選択され，またリンクされたコントロールが選択されることによって，ラベルもコントロールもハイライトします。

ステータスライン

TStatusLine クラスは，各種のステータス情報やヒント（ヘルプ）を表示するためのもので，通常，画面の最下段に表示されます。ステータスラインは，画面幅いっぱいまでとることのできる1行分の（1文字分の高さ）矩形領域です。このクラスは，アプリケーションを実行している最中に発生するイベントに応じて，動的な表示を行ないます。ステータスライン上の各項目は，***TLabel*** クラスと同様に，マウスかホットキーで選択することができます。たいていのアプリケーションクラスは，まず最初に，***TMenuBar***，***TDeskTop***，***TStatusLine*** の各オブジェクトを持つことになります。***TStatusLine*** に追加されたデータメンバは，***items*** と ***defs*** であり，いずれもポインタです。

items は，***TStatusItem*** オブジェクトからなるリンクリストへのポインタであり，現在使用中のリンクリストをポイントします。このリンクリストには，表示される文字列，対応するホットキー，およびリンクされる ***command*** ワードが格納されます。***defs*** データメンバは，***TStantusDef*** オブジェクトを要素とするリンクリストへのポインタであり，現在有効なヘルプコンテキストを示します。これを用いて，簡単なヒントを表示することができます。***TStatusLine*** は，専用の ***TApplication::initStatusLine*** を使用して，インスタンス化と初期化を行ないます。

ストリーム

ストリームは，入出力処理を行なう汎用のクラスです。従来のデバイスやファイル I/O においては，異なるデータ型のデータを読み出したり，変換したりするには，それぞれの関数グループを用意する必要がありました。これに対し，Turbo Vision のストリームでは，オーバーロード演算子 >> と << を使用して多態的な I/O 関数を作成することができます。オーバーロード演算子は，どのようなストリームデータを処理すべきかを知っていて，それに合わせて処理を行ないます。

Turbo Vision は，***TStreamableClass*** クラスを基本にした一般的なストリームマネージャを用意しています。このマネージャを使用することにより，ストリームとの間でのオブジェクトの読み書きを，型の対応を正しくとりながら行なうことができます。ストリーム入出力可能なすべてのクラスで定義されている関数がいくつかあります。ストリームマネージャは，それらの関数を使用して，そのクラスのオブジェクトの読み書きを行ないます。したがって，ストリームマネージャを使用するアプリケーションには，それらの関数を必ずリンクする必要があります。リンクするには，以下のようにして，***_ _link*** マクロを使って呼び出します。

```
_ _link(RClassName...);
```

これは，***TClassName*** 型のオブジェクトをストリームとの間で読み書きを行ないたいという意味です。これは，ストリームアプリケーションにクラスを**登録する**プロセスに他なりません。実際このことを**ストリーム登録**といいます。登録されたクラスは，そのクラスが必要とする別のクラスの登録も行なってくれるので，依存関係は自動的に処理されます。

オブジェクトのストリームは，**iostream** で使用される C++ 標準の **streambuf** クラスを使用して行なわれます。したがって，**iostream** の導出クラスとしてモデムとの間で読み書きを行なうクラスを作成すれば，モデムを介してオブジェクトの送受信を行なうこともできるということになります。

コレクション

TCollection は，一般的な要素集合を作成するためのクラスであり，いろいろな型の任意のクラスをその集合として扱うことができます。非 OOP 型言語に見られる配列，集合，およびリストとは異なり，Turbo Vision のコレクションでは，データサイズを動的に変化させることができます。***TCollection*** は，***TSortedCollection*** などのような，専門化したコレクションを作成するための基本シードクラスです。おもなデータメンバとしては，要素のコレクションへの汎用の（void**）ポインタである ***items*** があります。***TCollection*** には，インデックス付け，挿入，削除を行なう各メンバ関数の他に，繰り返し処理を行なうルーチンがいくつか用意されています。条件を指定するためのテスト関数をユーザ側で用意することによって，コレクション内でその条件に合致する一番最初の要素もしくは一番最後の要素をスキャンさせることもできます。また，***forEach*** メンバ関数を利用すれば，コレクション内の各要素に対して，ユーザ定義の動作を起こさせることもできます。

注意　***TNSCollestion*** は，***TCollection*** の基本クラスですが，***TCollection*** を使用したほうがよいでしょう。ここで，***TCollection*** に焦点を合わせたのは，そのためです。これは，***TNSSortedCollection*** と ***TSortedCollection*** の場合も同じです。

ソートされたコレクション

TSortedCollection は，キーを用いてソートしたコレクションを作成するための抽象クラスです。ソーティングは，純粋仮想関数である ***compare*** によって定義されます。したがって，導出されたそれぞれのクラスでは，いかなるクラスのコレクションに対しても，それぞれの目的に応じたソート法を指定することができます。***insert*** メンバ関数は，指定されたソート法に従いながら要素を追加していきます。またバイナリサーチを行なう ***search*** メンバ関数を使用して，キーの位置付けを高速に行なうことができます。***Boolean*** 型のデータメンバ ***duplicates*** が ***True*** の場合には，キー

を重複させることができます。

文字列コレクション

TStringCollection は，***TSortedCollection*** の拡張版であり，文字列からなるコレクションのソート処理を行なうものです。要は，***compare*** メンバ関数をオーバーライドして，アルファベット順にソートさせるようにしているだけです。***freeItem*** メンバ関数は，指定された文字列要素をコレクションから削除します。

リソース

リソースファイルは特殊なストリームであり，汎用のクラス(“要素”）を文字列キーでインデックス付けするものです。***TResourceFile*** は，***TStream*** からリソースファイルを導出するのではなく，***stream*** データメンバによって，ストリームにリソースファイルをリンクします。リソース要素は，***get*** (*key*) 呼び出しでアクセスされます(*key* は，インデックス用の文字列)。その他，メンバ関数として，***put*** (要素を指定されたキーを付けて格納する)，***keyAt*** (指定された要素に対するインデックスを取得する)，***flush*** (更新された内容をすべてストリームに書き出す)，***remove*** (キーで指定された要素を削除する)，***count*** (ファイル内の要素数を返す) があります。

リソースコレクション

TResourceCollection は，リソースファイルが使用するソート済みのリソースインデックスを要素とするコレクションです。リソースを処理できるように，***TStringCollection***，***freeItem***，および ***keyOf*** の各メンバ関数は，すべてがオーバーライドされます。

文字列リスト

TStringList は，特殊な文字列リソースを作成するためのものです。***get*** メンバ関数によって，文字列を数値インデックスでアクセスすることができます。***TStringList*** を利用すれば，簡単に多国語化が図れ，複数の国語を扱うテキスト処理アプリケーションを実現することができます。文字列リストは，ストリームマネージャを使用して，ストリームから読み出すことができます。文字列リストの作成や拡張には，***TStrListMaker*** を使用します。

TStringList では，数値でインデックス付けした作成済みの文字列リストに対してしかアクセスできません。***TStrListMaker*** は，文字列リストに文字列を追加するための ***put*** メンバ関数，およびストリーム入出力用のオーバーライド演算子 << および >> を用意しています。

第4章
ビュー

第1章，第2章を読み終えた方なら，Turbo Vision のアプリケーションが外から見るかぎりどんなものなのか，もうおわかりになったと思います。でもそのような外見の裏側には何があるのでしょうか。それが，第4章および第5章のテーマです。

“コントロールはテレビからもってきました”

Turbo Vision を使用する場合，これまでとは違って，画面に直接書き出さないようにしなければなりません。ユーザに対して何かを表示する場合，printf や puts などの stdlib 関数を使用するのではなく，Turbo Vision にそれを渡してやるようにします。そうすれば，Turbo Vision がその情報をしかるべき場所にしかるべきタイミングで表示してくれます。

Turbo Vision のアプリケーションにおいて，基本的なビルディングブロックとなるのが**ビュー**です。ビューは，画面上の矩形領域を管理するオブジェクトです。たとえば，画面の最上段にあるメニューバーはビューです。プログラムがその領域で行なう作業（たとえば，メニューバー上でのマウスクリック）は，すべてその領域を管理するビューによって処理されます。

メニューも，ウィンドウも，ステータスラインも，ボタンも，スクロールバーも，ダイアログボックスもビューです。さらに1行だけのテキストであっても一般にはビューなのです。Turbo Vision のプログラムで画面に表示されるものは，すべて一般にビューで**なければなりません**。また，ビューの性質でもっとも重要なことは，ビューが自分自身を画面に表示する方法を知っているいうことです。したがって，たとえばメニューシステムを作りたい場合には，これこれしかじかのメニューを内容とするメニューバーを作りたいという旨を Turbo Vision に知らせます。そうすれば，後は Turbo Vision が処理してくれるわけです。

おそらくビューだったとは思いもよらなかったでしょうが，もっともよく見かけるビューの例が，プログラムそのものです。プログラムは，画面全体を管理するビューなのです。ただ，ユーザとの対話を行なうために他のビューを（サブビューとして）どんどん設置していきますので，プログラムがビューであるとは気付かないだけのことです。また，後で説明しますが，ユーザから見て1個のオブジェクトに見えるもの（たとえばウィンドウ）も，多くの場合，関連するビューを組み合わせたグループとしてできあがっています。

単純なビューオブジェクト

99ページに示した階層図からわかるように，Turbo Vision のすべてのビューが，その上位オブジェクト（祖先）に ***TObject*** を持っています。***TObject*** には，単にすべてのオブジェクトに対する共通の祖先ということ以上の意味が少しあります。事実，Turbo Vision 自体は ***TView*** から始まります。

TView オブジェクト自体は，画面のまっさらな矩形領域として表されるだけのものです。プロトタイピングが目的で，画面のまっさらな矩形領域を作り出したいということでもなければ，***TView*** 自体をインスタンス化する意味はほとんどありません。しかし，外見上は単純なものであったとしても，***TView*** は，基本的な画面管理を行なう上で必要とされるメンバ関数やデータメンバを，すべて備えています。

TView から導出されたオブジェクトなら，できなければならないことが2つあります。

1. いつでも自分自身を表示できなければならないということです。***TView*** では，***draw*** という仮想メンバ関数が定義されており，したがって ***TView*** から導出されたオブジェクトも，それぞれ ***draw*** メンバ関数を備えています。これは大切なことです。というのも，ビューが別のビューでおおい隠されてしまったり，ビューの上に別のビューが部分的に重なったりすることはよくあることで，重なっていたビューが消されたり，移動したりした場合には，下になっていたビューは，それまで隠されていた部分を表示しなければならないからです。
2. 入ってきたイベントは，すべて処理できなければならないということです。第1章で述べたように，Turbo Vision のプログラムはイベントによって駆動されます。つまり Turbo Vision は，ユーザからの入力をいってに引き受け，その入力をアプリケーションのしかるべきオブジェクトに配送するわけです。ビューは，イベントが自分に関係するものであった場合に，何を行なえばよいかを知っておく必要があります。イベント処理については，第5章で説明します。

まずは，基本的なオブジェクトについて

ビューオブジェクトが何を**行なう**のかについて説明する前に，それは**何なのか**，画面上に自分自身をどうやって表示するのか，について少し触れておきます。

TPointについては，次の節で説明します。

ビューの位置は，2点で決定されます。つまり，左上隅（ビューの原点）と右下隅の点です。この2点は，クラスの中で ***TPoint*** 型のデータメンバとして表わされます。***origin*** データメンバは，ビューの原点を示す ***TPoint*** であり，***size*** データメンバは，右下隅を表わします。

ここで注意していただきたいことは，***origin*** は，オーナービューの座標系における点であるということです。たとえばデスクトップ上にウィンドウをオープンしたとすると，その ***origin*** データメンバは，デスクトップの原点を基準にしたウィンドウの *x*-*y* 座標を示します。これに対して，***size*** データメンバは，そのオブジェクト自身の原点を基準にした点です。***size*** は，原点から右下隅までの距離がどれだけあるかを教えてくれるわけですが，ビューの原点が別のビュー内のどこに位置しているかがわからなければ，右下隅の座標もわからないことになります。

TPointの取得

TPoint 型は，**単純そのもの**です。*x* および *y* という2つのデータメンバがあるだけで，***TPoint*** オブジェクトの座標を示します。メンバ関数としては，オーバーロード演算子＋＝と－＝の2つがあり，これらの演算子を以下のように使用することにより，*x* と *y* 座標の加減算を行なうことができます。

```
TPoint& operator +=( const TPoint& adder );

TPoint& operator -=( const TPoint& subber );
```

したがって，*p1* と *p2* が，それぞれ（*x1*,*y2*）と（*x2*,*y2*）の座標を持つ場合には，以下のようにすることができます。

```
p1 += p2;                // この演算の結果, p1 の座標は (x1 + x2, y1 + y2)
p1 -= p2;                // この演算の結果, p1 の座標は (x1 - x2, y1 - y2)
```

また ***TPoint*** 型には，－，＋，＝＝および！＝の4つのフレンド演算子もあり，以下のように引数に2個の点をとり，字義どおりの演算を行ないます。

```
TPoint p1(1,2), p2(3,4), p3;

p3 = p1 + p2;              // この演算の結果, p3 の座標は (4,6)
p3 = p2 - p1;              // この演算の結果, p3 の座標は (2,2)
if (p1 == p2) { // 点が一致したときの処理 }
if (p3 != p1) { // 点が一致しなかったときの処理 }
```

つまり，***TPoint*** クラスを使用することにより，ビューの座標を1個のデータメンバとして指定したり扱ったりすることができます。

TRect の導入

TPoint オブジェクトは，それだけで用いるには不便ですので，Turbo Vision では，直接使用することはあまりありません。原点とサイズは，どのビューにも備わっているので，通常は，いっしょにして ***TRect*** というクラスにして扱われます。***TRect*** は，***a*** と ***b*** の2つのデータメンバを持ちます。いずれも ***TPoint*** です。ビューオブジェクトの境界を指定するときには，***TRect*** のコンストラクタにその境界を渡してやります。

TRect と ***TView*** は，いずれも，ビューのサイズに関する便利なメンバ関数を用意しています。たとえば，あるウィンドウの内部にぴったりはまるビューを作成したい場合，ウィンドウにそのサイズを教えてもらい，そのサイズを少し小さめにして，内部に入れられるビューにはそのサイズを渡してやります。

ThisWindow と InsideView は，この例題用として作成したものです。

```
void ThisWindow::makeInside()
{
   TRect r = getExtent();      // r を ThisWindow のサイズに設定
   InsideView*inside;
   r.grow(-1, -1);        // 矩形領域を x と y の両方向に 1 だけ縮めます
   inside = new InsideView(r);    // 内部ビューを作成
   insert(inside);      // ウィンドウに新しいビューを挿入
// inside のポインタがとくに必要ない場合には,
// insert(new InsideView(r)); とすることもできます
}
```

getExtent は，***TView*** のメンバ関数で，そのビュー全体を覆う矩形の座標を引数 ***TRect*** にセットします。***grow*** は，***TRect*** のメンバ関数で，矩形の縦方向および横方向のサイズを拡大（パラメータが負の場合は縮小）します。

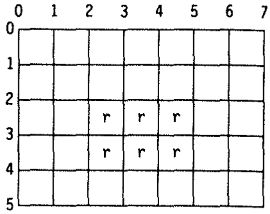
Turbo Vision の座標

Turbo Vision のメソッドでの座標の与え方は，従来のやり方とは異なります。多くの場合，座標系を画面上の文字数で指定しますが，Turbo Vision では，座標を文字の**升目の**点で指定します。

たとえば，***r*** が ***TRect*** オブジェクトであるとき，r.assign(0,0,0,0) は，サイズが 0 の矩形，すなわち単なる点を意味します。何かを入れることのできる最小の矩形を作るには，r.assign(0,0,1,1) とします。

図 4.1 は，r.assign(2,2,5,4) で作成した ***TRect*** を表わしています。

図 4.1
Turbo Vision の座標系



図にあるように，r.assign(2,2,5,4) は，6 文字分の大きさの矩形を作ります。この座標系は少し不便な点もありますが，矩形のサイズや隣接する矩形の座標などを**非常に簡単に**計算することができます。

ビューの形態

ビューオブジェクトの形態は，ビュー自身の ***draw*** メンバ関数によって決定されます。ビュー同士は，一般に，その形態で区別されるわけですから，***TView*** から導出されたほとんどすべてのクラスに，それ自身の ***draw*** を用意する必要があります。

形態に関して，すべてのビューに当てはまる規則があります。すなわち，ビューは，

■それが管理する領域全体をサポートしなければならず，また，
■いつでも，自分自身を表示できなければならない，

ということです。この2つの性質は，非常に重要な意味を持っていますので，これについて，少し説明しておきましょう。

守備範囲

ビューがその守備範囲について責任を持たなければならないのには，それなりの理由があります。ビューには，画面上の矩形領域が割り当てられます。もしビューがその守備範囲全体を管理しないとすれば，管理されない領域の内容は，未定義だということになります。何でもかんでもそこに表示されうることになり，その領域については，まったく制御できないことになります。第2章のサンプルプログラム TVGUID06.CPP は，もしビューがその形態の一部を自分の手で管理しないとしたら，どういうことになるかを示してくれるプログラムです。

要求に基づく表示

ビューは，また，**いつでも**自分自身を画面に表示できなければなりません。というのは，ビューの一部が別のビューによって隠され，後で重なっていたビューが，あるいはそのビュー自体が移動することもあるからです。いずれの場合でも，呼び出されたときにちゃんと自分自身を表示できるように，ビューは現在の状態をいつでも把

握しておく必要があります。

このことは，ビューが何も行なわないこともありうるということを意味しています。たとえば，ビューが完全に隠れてしまっているとか，画面上にないとか，ビューを管理下に置いているウィンドウが，ビューがまったく見えなくなるぐらいまで縮小してしまったとかいう場合です。このようなことが起こった場合の処理は，たいてい自動的に行なわれますが，ビューを作成する場合には，ビューがいつでも自分自身を表示できるようにしなければならない，ということを憶えておいてください。

この点は，Turbo Vision 以外の多くのウィンドウシステムと異なる点です。そのようなシステムでは，たとえばウィンドウ内での表示は固定的なものとなっています。つまり，ウィンドウに表示してしまえば，何かがその上にかぶさり，後でそれが取り除かれたとしても，もとどおり残っています。Turbo Vision の場合は，ビューの上にかぶさっていたものを取り除いたとき，ビューがもとどおりに表示されるとは限りません。何かに覆われているときに，そのビューに対する変更がなされることもあるからです。

ふさわしい装いをする

ビューの動作は，ほとんど ***handleEvent*** というメンバ関数によって決定されます。***handleEvent*** は，イベントオブジェクトを渡されますが，その処理方法は 2 つに分かれます。すなわち，イベントに従って何らかの処理を行ない，処理済みの印をつけるか，さもなければ，そのイベントを処理すべき次のビューに渡してやる（そのようなビューがあった場合）かのいずれかです。

イベント処理については，第 5 章『イベント駆動型プログラミング』で詳細に説明します。

ビューの動作で重要なことは，イベントに対してどのような対応をするかということです。たとえば，ウィンドウが ***cmClose*** コマンドのイベントを受け取った場合，普通は，そのウィンドウをクローズします。***cmClose*** コマンドに対して何か別の対応をさせるようにすることもできるでしょうが，適当ではありません。

複合ビュー

TView の直系下位のオブジェクトのうち，もっとも重要な働きをする ***TGroup*** については，すでにいくらかおわかりのことと思います。***TGroup*** およびその導出クラスは，まとめて，グループと呼ばれるものです。***TGroup*** から導出されていないビューは，ターミナルビューと呼ばれます。

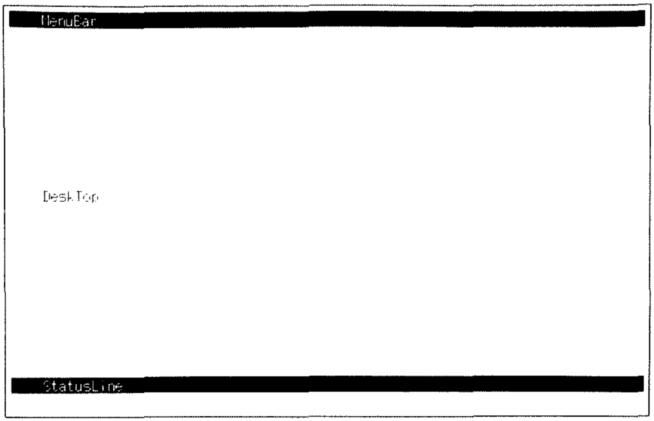
グループは，基本的には，他のビューを管理下に置くただの空の箱にすぎません。内部的には，グループはビューであり，したがってビューが行なうべきことは，すべて責任を持って行なわなければなりません。たとえば，画面上の矩形領域を管理するとか，必要なときに自分自身を表示するとか，その守備範囲内で起こったイベントを処理するといったことです。違いは，それを**どのように**処理するかという点です。ほとんどのことは，**サブビュー**に処理してもらいます。

グループとサブビュー

サブビューとは，別のビューの管理下に置かれるビューのことです。つまり，あるビュー（グループ）が，画面上のその守備範囲の一部を，別のビュー，すなわちサブビューに任せたということです。以後は，サブビューがその領域を管理します。

うってつけの例が ***TApplication*** です。***TApplication*** は，画面上の領域，この場合は画面全体，を管理するビューですが，メニューバー，デスクトップ，ステータスラインという 3 つのサブビューを従えるグループでもあります。***TApplication*** は，画面上の領域を，この 3 つのサブビューに任せるわけです。メニューバーは最上段の 1 行を，ステータスラインは再下段の 1 行を，デスクトップは，その間にあるすべての行を獲得します。**図 4.2** は，一般的な ***TApplication*** の画面を示しています。

図 4.2
TApplication の画面配置



アプリケーション自体が，画面に現われることはありません。したがって，アプリケーションは目には**見えません**。アプリケーションの形態は，すべてそれが従えているビューによって決まるのです。

グループの導入

サブビューをグループに組み入れるにはどうすればよいのでしょうか。組み入れることを挿入といいます。サブビューは，作成したら挿入すればよいのです。先の例をとりあげるなら，***TApplication*** コンストラクタが，以下の疑似コードにあるように，3つのオブジェクトを作成し，それをアプリケーションに挿入します。

```
createStatusLine( cStatusLine ),
createMenuBar( cMenuBar ),
createDeskTop( cDeskTop )
if createDeskTop != 0  insert(deskTop);
if createStatusLine != 0  insert(statusLine);
if createMenuBar != 0  insert(menuBar);
```

新たに作成されたビューは，挿入が完了してしまいさえすれば，グループの一員となります。上の例では，***TApplication*** は，その領域を3つの部分に分けて，サブビューにそれぞれ1つずつを任せています。この例では，サブビューがまったくオーバーラップしていないので，視覚的に非常にわかりやすいものとなっています。

しかし，ビューがオーバーラップしてはならないという法はあり

ません。事実，ウィンドウ環境がもたらす特典の1つとして，デスクトップ上にウィンドウを何枚もオーバーラップさせることができるということがあります。幸い，グループは（デスクトップも含めて），オーバーラップするサブビューを処理できるようになっています。

グループは，サブビューを挿入した順番を記録します。その順番のことを**Zオーダー**といいます。Zオーダーは，サブビューを表示する順番およびサブビューにイベントを渡す順番を決めるものです。

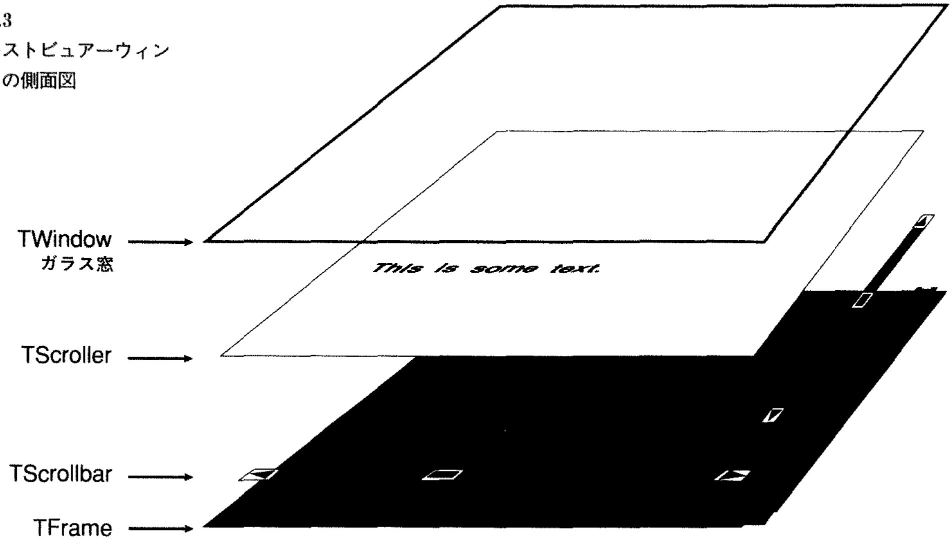
別の角度から捉えたZオーダー

Zオーダーという用語には，サブビュー同士の関係を3次元空間で表現するということの意味合いがあります。前述したように，すべてのビューが，そのビューの平面上における位置と大きさを持っており（x 次元，y 次元），***origin*** フィールドおよび ***size*** データメンバで決定されています。しかし，ビューおよびサブビューはオーバラップすることができるので，どのビューがどのビューの手前にあるかを Turbo Vision が把握しておくためには，第3次元として，Z次元を加える必要があります。

そこで，ユーザから見てもっとも手前にあるビューから始めて，画面の“奥の方向に”進んでいくときに出会うビューの順序のことをZオーダーということにします。一番最後に挿入されたビューが“手前の”ビューです。

したがって，画面は，その上に何かが表示される平面というよりは，ビューの3次元の世界への窓口を意味するガラス窓だと考えたほうがよいでしょう。**図4.3**を見ればわかるように，なるほどグループはビューをサンドイッチにしたものと考えることができるでしょう。

図 4.3
テキストビュアーウィンドウの側面図



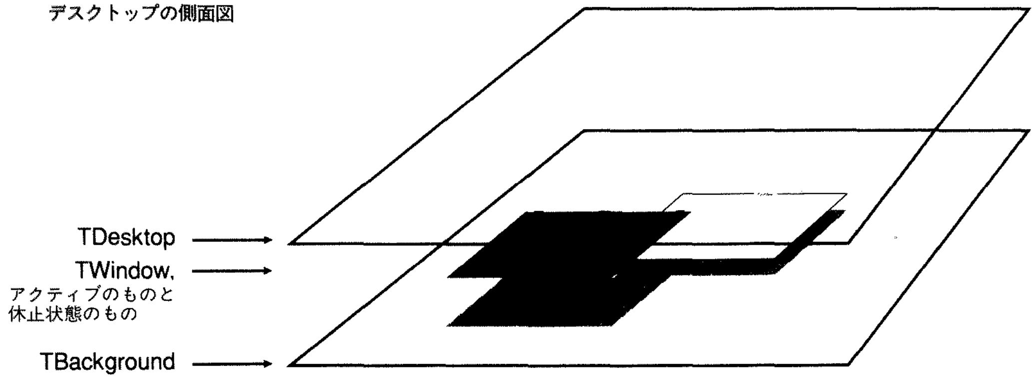
ウィンドウ自体は，ビューのグループにかぶさる単なるガラス窓です。そこに見えるのは，画面のガラスの裏側に投影されたビューだけですから，ビュー同士がオーバーラップしないかぎり，どのビューが手前にあるのかはわかりません。

通常，ウィンドウにはフレームが付けられますが，フレームは，どのサブビューよりも先に挿入されます。つまり，フレームは“背景”となるビューです。スクロールインテリアを作成するのであれば，スクロールバーがフレームの上に置かれます。全体を手前から見ると，スクロールバーはフレームの一部であるように見えますが，側面から見ると，実際にはフレームの一部を覆うようにしてフレームの“上に”スクロールバーが浮いていることになります。

最後に，フレームの境界から内側全体を覆うようにして，スクローラ（スクロールオブジェクト）が挿入されます。テキストは，スクローラ上に書かれます。ウィンドウを通して見るとウィンドウ上に書かれるように見えますが，そうではありません。

もっと大きな範囲で捉えてみると，デスクトップも単なる大きなガラス窓でしかなく，大きなサンドイッチを覆っていることがわかります。さらにそのサンドイッチの中身も，多くは，小さなサンドイッチからなっていることがわかります。この様子を**図 4.4** に示します。

図 4.4
デスクトップの側面図



グループ（この場合はデスクトップ）もまた，ガラス窓です。この場合，一番最初に挿入されるサブビューは ***TBackground*** オブジェクトであり，他のすべてのビューの“背景”に置かれます。この図では，デスクトップ上に，スクロール用のインテリアビューを備えたウィンドウが置かれています。

グループの概要

ビューは自分自身で表示する方法を知っていなければならないという規則に関しては，グループは例外となっています。というのは，グループ**自体**が，自分自身を表示することはないからです。そのかわり，***TGroup*** は，そのサブビューにサブビュー自身の表示を行なわせます。

サブビューが呼び出された場合は，Z オーダーで表示されます。つまり，グループに最初に挿入された（もっとも奥にある）サブビューが，最初に表示されます。したがって，サブビューがオーバーラップする場合，一番最後に挿入されたサブビューが一番手前にくることになります。

グループ下の各サブビューは，そのグループが管理する領域全体を，協調しながらサポートしなければなりません。たとえば，ダイアログボックスはグループですが，そのサブビューであるフレーム，インテリア，コントロール，および固定文字列は，協力しながら，ダイアログボックスビューの領域全体を“完全に”サポートする必

要があります。そうしなければ，ダイアログボックスに“穴”ができてしまい，予期せぬ結果を招くことになります。

グループ内のサブビューが自分自身の表示を行なう場合，その表示は，自動的にそのグループの境界でクリップされます。サブビューがクリップされるわけですから，ビューを初期化してグループに挿入する場合には，そのビューの少なくとも一部をグループの境界内に残しておく必要があります（たとえば，あるウィンドウを捉えてそのコーナーが1つしか見えないようになるまでデスクトップの外に追い出すこともできますが，いずれにせよそのウィンドウをまた後で使えるようにするためには，どこか一部をデスクトップ上に残しておく必要があります)。その結果，グループの境界内に残っているサブビューの一部しか見えなくなります。

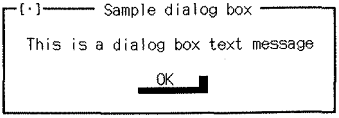
デスクトップが ***TGroup*** オブジェクトなのなら，その背景はどこにあるのだろうかと，考えた方もいらっしゃるのではないでしょうか。デスクトップは，初期化時に ***TBackGround*** 型のサブビューを作成し管理下におきます。***TBackGround*** は，画面全体の背景を同じパターンで埋め尽くすためだけのビューです。背景は，最初に挿入されるサブビューなので，その上に他のビューが表示され，重ねられていきます。

ビュー間の関係

ビュー同士の関係は，2通りの方法で捉えることができます。1つは，Turbo Vision のクラス階層中のオブジェクト同士として，もう1つは，ビューツリーのメンバ同士としてです。この違いは，よく理解しておく必要があります。

図 4.5 の簡単なダイアログボックスを例にとってみましょう。そこには，フレームとテキストメッセージが1行，それにダイアログボックスをクローズするためのボタンが1個付いています。ビューツリーの用語を使うなら，このダイアログボックスは，***TFrame*** と ***TStaticText*** と ***TButton*** を1個ずつ備える ***TDialog*** ビューだということになります。

図 4.5
簡単な
ダイアログボックス



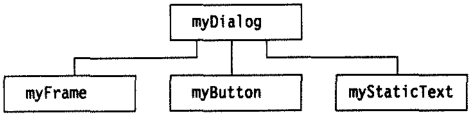
クラス階層

1つの見方として，ビュー同士は，クラス階層内で基本クラスと導出クラスの関係にあります。100ページの階層図にあるように，***TButton*** は，***TView*** の導出クラスで，実際 ***TButton*** は ***TView*** **なのです**が，それをボタンならしめるためのメンバが追加されています。***TDialog*** も ***TView*** の導出クラスです（***TGroup*** および ***TWidow*** を介して)。したがって，***TButton*** と共通するものをたくさん備えています。***TButton*** と ***TDialog*** は，Turbo Vision の階層において，遠い“親戚”同士となっています。

所有関係

もう1つの見方として，ビュー同士はビューツリー上で関係づけることができます。ビューツリー図(**図 4.6**)にあるように，*myDialog* という ***TDialog*** オブジェクトは，*myButton* という ***TButton*** を所有します。ここでいう関係とは，クラスの間のものではなく（***TDialog*** は，***TButton*** の基本クラスでは**ありません**)，オブジェクト（クラスのインスタンス）間の関係，すなわちオーナーとサブビューとの関係です。

図 4.6
ダイアログボックスの
簡単なビューツリー



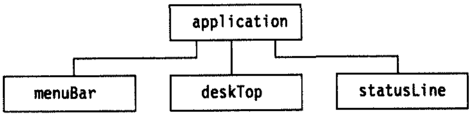
実際のプログラムでは，*myButton* オブジェクトとビューツリーにおけるそのオーナー(*myDisalog* オブジェクト)とを相互動作させる必要がでてきます。同時に，*myBotton* は，その最終的な祖先(***TView***) から継承した属性にしたがって表示を行なうことになります。この所有関係と継承という2つの関係を混乱しないようにしてください。

Turbo Vision のアプリケーションを実行する過程は，ビューが他のビューをインスタンス化しながら所有していくという意味で，ツリーに似ています。アプリケーションがウィンドウをオープンしたり，クローズしたりするのに伴い，オブジェクトのインスタンスが挿入されたり取り除かれたりすることから，ビューツリーは大きくなったり小さくなったりします。もちろんクラス階層に関しては，標準的なクラスから新たなクラスを導出するのに伴い，大きくなる一方です。

サブビューとビューツリー

以前にも述べましたが，***TApplication*** グループビューは，作成し，挿入した3つのサブビューを，所有・管理します。これは，ビューツリーを作る関係であると見なすことができます。***TApplication*** の静的メンバである ***application,menuBar,deskTop,*** および ***statusLine*** が，これらのオブジェクトへのポインタとなっています（説明を簡潔にするために，よく ***application*** をメインプログラムのオブジェクトであるかのように使っていますが，厳密にいうならメインプログラムのオブジェクトは，****application*** です)。**図 4.7** にあるように，***application*** を幹とし，***menuBar,deskTop*** および ***statusLine*** を枝として捉えることができます。***TApplication*** クラスのすべてのオブジェクトで，静的データメンバ1個につき1個のインスタンスしか存在しないということを忘れないでください。

図 4.7
Turbo Vision における基本的なビューツリー



繰り返しますが，**図 4.7** に示した関係は，クラス階層では**なく**，データ構造のモデルです。この図におけるオブジェクト間のリンクは，**所有関係**を示すものであり，継承を示すものではありません。

オーナーとサブビュー

通常のアプリケーションでは，ユーザがマウスをクリックするこ

とによって，あるいはキーボードから入力することによって，ビューをふやしていくことになります。これらのビューは，通常，デスクトップ上に現われ，ツリーの枝をさらに伸ばすことになります。

ビューの形態と動作は，そのビューがどのオーナーに属しているかに大きく左右されますので，オーナーとサブビューとの間のこれらの関係を理解することは，非常に重要なことです。

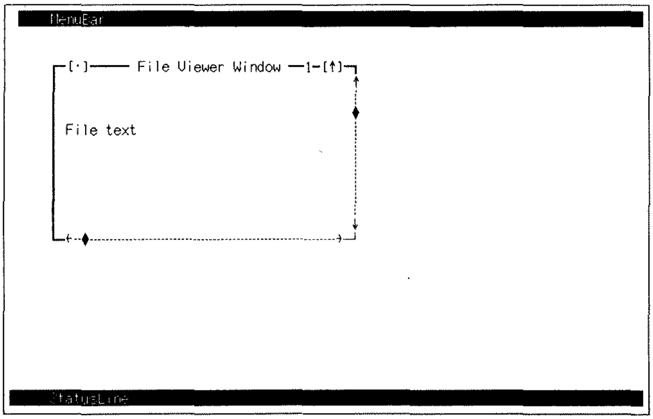
ツリーを広げていくプロセスを追ってみましょう。たとえば，ユーザがファイルビュアーウィンドウを呼び出すメニュー選択項目をクリックしたものとしましょう。すると，ファイルビュアーウィンドウは，ビューになります。つまり Turbo Vision が，ウィンドウを作成し，それをデスクトップにリンクするわけです。

これと同様のオブジェクトが，少し違った形で，図 4.3 に示されています。

ウィンドウは，たいてい，いろいろなサブビューを備えているのが普通です。たとえば，***TFrame*** オブジェクト（ウィンドウを囲むフレーム），***TScroller*** オブジェクト（テキストの配列を管理し，それをスクロールさせることのできるインテリアビュー），2 つの ***TScrollbar*** オブジェクトです。ウィンドウを発生させると，ウィンドウは自分用のサブビューを作成し，保有・管理します。

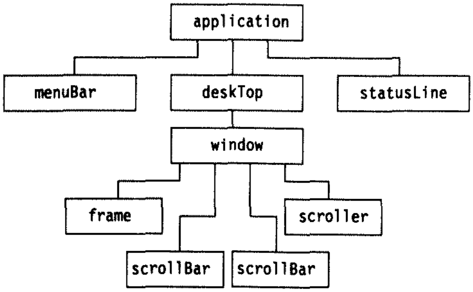
いろいろなビューをたて横付け加えた結果，アプリケーションは，**図 4.8** のようなものとなります。

図 4.8
デスクトップにファイルビュアーを付加したところ



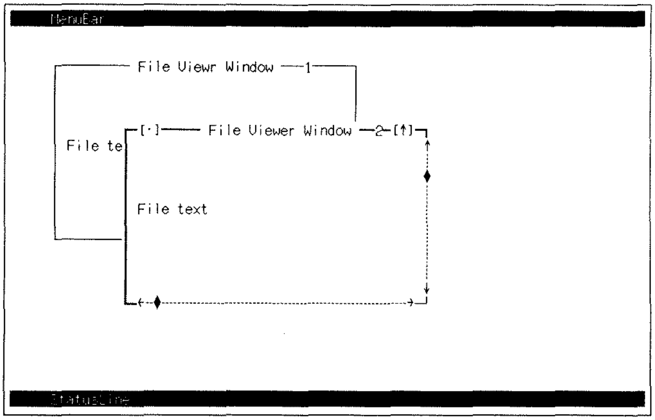
ビューツリーも，**図 4.9** のように，いくらか複雑になってきました（これは，所有関係のリンクを示すものであることをお忘れなく）。

図 4.9
ファイルビュアーを追加したときのビューツリー



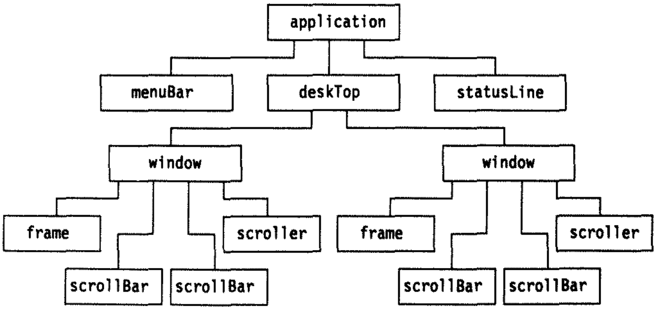
今度は，ユーザが同じメニュー選択項目をクリックして，ファイルビュアーウィンドウをもう 1 個作ったとしましょう。すると，Turbo Vision は，**図 4.10** にあるように，2 番目のウィンドウを作成し，デスクトップにリンクすることになります。

図 4.10
デスクトップにファイルビュアーを追加したところ



ビューツリーも，それに応じて，**図 4.11** に示すように，さらに複雑なものとなります（これも，所有関数のリンクを示しています）。

図 4.11
ファイルビュアーを 2 個追加したときのビューツリー



第5章で説明しますが，プログラムの制御フローは，このビューツリーに沿って下っていくことになります。上の例で，ファイルビュアーウィンドウのスクロールバーをクリックしたものとします。その場合，そのクリックは，どのようにしてちゃんとしたところに伝えられるのでしょうか。

アプリケーションオブジェクト（ユーザプログラム）は，マウスクリックがあったことを知ると，それがデスクトップの管理下にある領域内のものであることを認識し，デスクトップオブジェクトにそれを渡してやります。するとデスクトップは，そのクリックがファイルビュアーの管理下にある領域内のものであることを検出し，今度はファイルビュアーにそれを渡してやります。今度，ファイルビュアーは，そのクリックがスクロールバー上で行なわれたことを検出し，スクロールバーのビューにそのクリックに対する処理を行なわせ，しかるべき仕事が行なわれることになります。

イベントの渡っていく経路については，第5章で説明します。

この処理のフローについての内部的なメカニズムについては，ここでは重要ではありません。大切なことは，ビュー同士がどのように接続されるかという点です。構造的にどれだけ複雑なものになろうと，ビューは，すべて最終的にはアプリケーションオブジェクトに接続されるのです。

もしユーザが2番目のビュアーのクローズアイコンもしくはメニュー項目の Close Window をクリックしたら，2番目のファイルビュアーはクローズするわけですが，その場合，Turbo Vision は，ビューツリーからそのファイルビュアーを取り除き，廃棄します。ウィンドウは，管理下に置いていたサブビューをすべて消去した後，自分自身も消去します。

ユーザは，最終的には，ビューがもともと4つあった時点にまで戻ることになります。さらに，ある時点になれば，ユーザは作業を終了したとして ***GRPH-X*** を押すか，メニューの Exit を選択します。その場合，***TApplication*** オブジェクトは，管理下に置いていた3つのサブビューを消去した後，自分自身も消去します。

選択されているビューとフォーカスビュー

フォーカスの当たっているビューとは，選択されているビューのチェイン（チェインはアプリケーションから始まる）における最後のビューのことです。

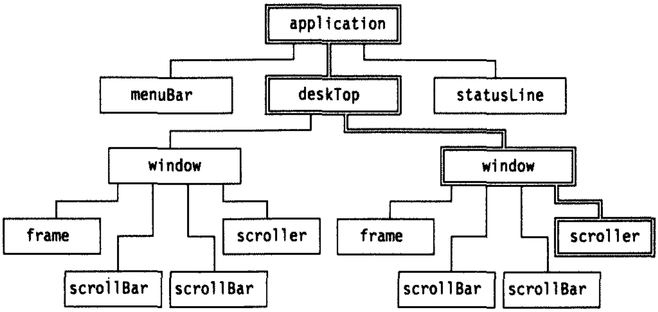
それぞれのビューグループ内では，いつでも1つのサブビューのみが**選択**されています。たとえば，メニューバーとデスクトップとステータスラインを作成し，挿入した直後は，デスクトップが選択されているビューです。なぜなら，デスクトップにおいて作業が進められることになるからです。

デスクトップにウィンドウを何個かオープンしたとすると，ユーザが現在作業を行なっているウィンドウが選択されているウィンドウです。このウィンドウのことを**アクティブな**ウィンドウともいいます（一般に，最前面にあるウィンドウのこと）。

アクティブなウィンドウの内部で，選択されているサブビューのことを**フォーカス**ビューといいます。フォーカスビューは，ユーザが注目しているビュー，あるいはこれから動作の起こされるビューであるということができます。エディタウィンドウの場合，テキストとその中のアクティブなカーソルを所有するインテリアビューがフォーカスビューとなり，ダイアログボックスの場合，ハイライトしているコントロールがフォーカスビューとなります。

図4.11に示したアプリケーションでは，***application***はモードつきビューであり，***deskTop***が選択されているビューです。デスクトップ内では，2番目（後で挿入されたほう）のウィンドウが選択され，アクティブになっています。そのウィンドウ内では，スクロールインテリアが選択されています。スクロールインテリアはターミナルビューですので(つまりグループではないので)，それがチェインの最後，すなわちフォーカスビューとなります。**図4.12**は，先ほどと同じビューツリーですが，フォーカスビューのチェインを二重線の箱で表わし，ビューツリー内の所有関係を示しています。

図 4.12
フォーカスチェイン



どれがフォーカスビューであるかを知ることで，キーボードからの入力を受け取るビューがどれであるかがわかります。詳細については，第5章『**イベント駆動型プログラミング**』のフォーカスイベントの節を参照してください。

フォーカスビューの検出

モノクロディスプレイの場合，Turbo Vision は大カッコ文字を使ってフォーカスを表わします。

フォーカスビューは，通常，画面上で何らかの形でハイライトされています。たとえば，デスクトップ上にウィンドウをいくつかオープンしている場合，アクティブなウィンドウは太いフレームで囲まれています。他のウィンドウのフレームは，細線で表示されます。ダイアログボックス内では，フォーカスコントロールは，他のコントロールより明るくされ(コントロールもビューです！)，それが**リターン**キーを押した場合に動作を起こすコントロールであることを示しています。したがって，フォーカスコントロールは，デフォルトのコントロールにもなっているわけです。

ビューがフォーカスを得るのはどんなときか？

ビューがフォーカスを得る場合としては，2通りあります。すなわち，ビューが作成されたときにデフォルトとしてフォーカスを得る場合，もしくはユーザが何らかの動作を行なった場合の2通りです。

グループが作成されるとき，オーナービューは，そのサブビュー

のうちどれにフォーカスを当てるかを，サブビューの ***select*** メンバ関数を呼び出すことで指定します。これによって，**デフォルトのフォーカス**が決まります。

ユーザが今とは別のビューにフォーカスを当てたい場合には，一般には，その別のほうのビューでマウスをクリックします。たとえば，デスクトップにウィンドウをいくつかオープンしている場合，別のウィンドウを選択するには，その別のほうのウィンドウでマウスをクリックするだけでよいのです。ダイアログボックスの場合には，***TAB*** を押すか，特定のビューでマウスをクリックするか，あるいはホットキーを押すことでフォーカスを移動させることができます。***TAB*** を押した場合には，フォーカスはすべての有効なビューを巡回していきます。

ビューの中には，デスクトップの背景とか，ウィンドウのフレームとか，スクロールバーなどのように，選択できないものもあります。ビューを作成する場合には，そのビューが選択可能かどうかを指定することができます。ビューは，その指定にしたがって，自分が選択可能であるかどうかを決定していきます。たとえば，ウィンドウのフレームをクリックした場合，***TFrame*** オブジェクトは選択することのできないオブジェクトなので，フレームがフォーカスを得ることはありません。つまり，フレームはフォーカスビューになることができないということを知っているわけです。

フォーカスチェイン

詳細については，第5章『イベント駆動型プログラミング』を参照してください。

メインとなるアプリケーションからスタートして，そこで選択されているサブビューを追っていき，さらにそれ以降の選択されているサブビューを追い続けていくと，ついにはフォーカスビューにまで到達することになります。***TApplication*** オブジェクトから始まり，フォーカスビューにまでつながるビューのチェインのことを，**フォーカスチェイン**といいます。フォーカスチェインは，キー入力などのイベントがフォーカスの当たっているビューに渡っていく場合の経路として利用されます。

モードつきビュー

モードとは，動作あるいは機能遂行の様式のことです。プログラムは，いろいろな動作モードを備えていますが，動作モードは，通常，いろいろな制御機能や制御の及ぶ範囲によって区別されます。たとえば，ボーランドのIDE（統合開発環境）には，編集，デバッグ，コンパイル，実行の各モードが備わっています。どのモードがアクティブであるかによって，キーボード上のキーの持つ意味が変わってきます（あるいは，全然意味を持たなくなることもある)。

Turbo Visionのビューでは，動作モードを定義できるようになっています。動作モードを定義したビューのことを**モードつきビュー**といいます。モードつきビューの典型的な例がダイアログボックスです。通常，あるダイアログボックスがアクティブであるときには，そのダイアログボックス以外のものは，何も機能しません。メニューやそのダイアログボックスに属さないコントロールは使用できないわけです。また，そのダイアログボックス外でマウスをクリックしても，何も起こりません。そのダイアログボックスは，それがクローズされるまで，プログラムの制御を握っているわけです(ダイアログボックスの中にも，たまにモードなしのものがありますが，それは例外です)。

ビューをインスタンス化しモードつきにした場合，そのビューおよびそれに属すサブビューだけが，ユーザとの対話を行なうことができます。モードつきビューは，プログラムの一部に対して“スコープ”を定義していると見ることもできます。モードつきビューは，そのモードつきビューにおいてどのような動作が有効なのかを決定するわけです。つまりイベントは，そのモードつきビューもしくはそれに属すサブビューによってしか処理されないのです。ビューツリーの中で，そのモードつきビューおよびそれに属すもの以外は，すべて休止状態となります。

ビューがモードつきであったとしても，ステータスラインは，常に“ホット（有効)”です。

しかし実際には，この約束にも例外が1つあります。それはステータスラインです。Turbo Visionでは，ちょっとした“裏口”が用意されており，ステータスラインは，いつでも働くようになっています。したがって，モードつきのダイアログボックスがステータスラ

インを従えているわけではないのに，モードつきのダイアログボックスを実行しているときでも，ステータスラインの項目はアクティブになっています。そして，ステータスラインが発行したイベントやコマンドは，モードつきビューが発行したかのごとく処理されます。

Turbo Vision のアプリケーションを実行しているときには，**常に**モードつきビューが存在します。プログラムの実行を開始したとき，あるいはプログラムを実行している最中でも，かなりの間，アプリケーションそのもの，すなわちビューツリーの先頭にある ***TApplication*** オブジェクトがモードつきビューとなっています。

デフォルト動作の変更

これまで見てきたものは，ほとんどが標準的なビューのデフォルトの動作でした。しかし場合によっては，ビューに少し違った立居振舞いをさせたいことがあり，実際 Turbo Vision ではそれをサポートしています。この節では，標準的なビューを変更する方法について説明します。

Turbo Vision のすべてのビューが，フラグとしてビットマップ化されたデータメンバを 4 つ備えており，それを使用してビューの動作をいろいろと制御できるようになっています。その 4 つのフラグのうち，***ushort*** 型の ***options*** フラグ（16ビット），***uchar*** 型の ***growMode*** フラグ(8ビット)，および ***uchar*** 型の ***dragMode*** フラグ(8ビット)の 3 つを，以下で説明します。4 つ目の ***ushort*** 型の ***eventMask*** フラグについては，第 5 章『**イベント駆動型プログラミング**』で説明します。

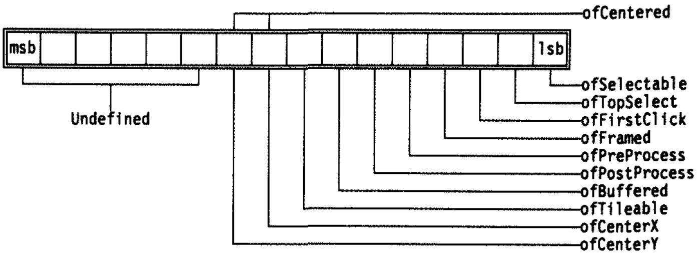
またその他に，ビューの現在の状態を表わす ***ushort*** 型の ***state*** メンバ（16ビット）もありますが，上の 4 つとは異なり，基本的に“リードオンリー”となっています。その値は，***setState*** メンバ関数でしか変更してはならないようになっています。詳細については156ページを参照してください。

options フラグ

options はビットマップフラグであり，すべてのビューに備わっています。***TView*** の導出クラスそれぞれで，デフォルト動作を決定するために ***options*** に設定されるビットは異なります。

options の各ビットは，**図 4.13** に示すように，ニーモニックで定義されています。各ビットフィールドの意味は以下のとおりです。

図 4.13
options の
各フラグビット



ofSelectable

このビットをセットした場合(1にした場合)，そのビューをマウスで選択することができます。ビューがグループ内のものであった場合には,マウスもしくは ***TAB*** キーでそのビューを選択することができます。ビューが，単に画面に何らかの情報を表示するためだけものであるなら，ユーザがそれを選択できないようにするのが一般的です。たとえば，固定文字列オブジェクトやウィンドウフレームは，通常，選択できないようになっています。

ofTopSelect

セットした場合，そのビューが選択されると，そのオーナーに属すサブビューの先頭に移されます。このオプションは，主として，デスクトップ上のウィンドウに対して使用します。このオプションを，グループ内のビューに使わないようにしてください。

ofFirstClick

セットした場合，ビューを選択するためのマウスクリックイベントが，このビューに送られるようにします。ボタンがクリックされた場合には，クリック1回でそのボタンの選択とそれに伴う動作を行なわせたいわけですから，ボタンの ***ofFirstClick*** はセットしておきます。しかしウィンドウをクリックした場合には，そのウインドウ自身を選択するのではなく，それとは別の処理をそのクリックに対応させたい場合があります。その場合には，このフラグをセットしないでおきます。

ofFramed

これをセットすると，ビューにはそれを囲むフレームが表示されます。たとえば，1個のウィンドウ内に複数の窓枠を設置したい場合に使用されます。

ofPreProcess

これをセットすると，そのビューは，フォーカスイベントが，フォーカスビューに渡る前に，そのイベントを処理することができます。詳細については第5章の179ページの節を参照してください。

ofPostProcess

これをセットすると，そのビューは，フォーカスイベントが，フォーカスビューに渡された後で，そのイベントを処理することができます。ただし，処理できるのは，フォーカスビューがそのイベントをクリアしなかった場合です。詳細については第5章の179ページの節を参照してください。

ofBuffered

このビットをセットすると，グループは画面への表示を高速に行なうことができます。グループが1回目にそれ自身を表示するよう要求されると，このビットがセットされていて，かつ充分なメモリが利用できる場合には，自動的にその表示内容をバッファにセーブします。そのグループが次回に自分自身の表示を要求された場合には，そのグループ下のサブビューそれぞれに自分自身の表示を行なわせていくかわりに，バッファにセーブされた表示内容を画面にコピーします。***new*** などのメモリ要求があったときにメモリを使いきっていた場合には，そのメモリ要求を満たすことができるようになるまで，Turbo Vision のメモリマネージャがグループのバッファを廃棄していきます。

グループにバッファがとられた場合，***TView::lock*** 呼び出しによ

って，そのグループの画面へのすべての出力を，***TView::unlock*** が呼び出されるまでやめさせることができます。***unlock*** が呼び出された時点で，グループのバッファが画面に書き出されます。ロックすることによって，画面への表示がいろいろと変更されるときに起こるどたばたを抑えることができます。たとえば，デスクトップがそのサブビューを表示していく場合には，タイル表示（タイルのように並べていく）の場合でもカスケード表示（どんどん重ねて表示する）の場合でも，自分自身をロックします。

ofTileable

セットした場合には，デスクトップが，オープンしているウィンドウを表示する際に，タイル表示することも，カスケード表示することもできます。ウィンドウをタイル表示したくない場合には，このビットをクリアしなければなりません。その結果，このようなウィンドウは同じ位置にとどまり，タイル表示可能なウィンドウは，自動的にタイル表示されていきます。

ビューをタイル表示するか，カスケード表示するかは，***TApplication::handleEvent*** で以下のようにして簡単に指定することができます。

この例は，マスターディスクに入っているFVIEWER.CPPの一部です。

```
switch( event.message.command )
{
    ...
    case cmTile:
        tile();
        break;
    case cmCascade:
        cascade();
        break;
    ...
}
clearEvent( event );
```

ビューが多すぎて，うまくカスケード表示できない場合には，デスクトップは何もしません。

ofCenterX

セットすれば，ビューをグループに挿入する際，そのビューを水平方向（ x 方向）でセンタリングします。

ofCenterY

ビューをグループに挿入する際，そのビューを垂直方向（ y 方向）でセンタリングします。テキストモードが25行（ノーマルモード）でも31行（ハイレゾモード）でも，ウィンドウを不自由なく操作できる位置に配置するために，このビットを利用することがあります。

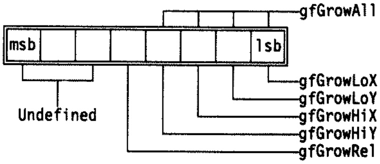
ofCentered

ビューをグループに挿入する際に，そのビューを水平，垂直の両方向でセンタリングします。

growMode フラグ

growMode データメンバは，そのビューのオーナーであるグループに対してリサイズが行われたとき，そのビューがどのような変更を受けるかを決定します。***growMode*** の各ビットの定義は以下のとおりです。

図 4.14
growMode の各フラグビット



gfGrowLoX

これがセットされた場合，そのビューの左辺とオーナーの左辺との距離が一定に保たれます。

gfGrowLoY

これがセットされた場合，そのビューの上辺とオーナーの上辺との距離が一定に保たれます。

gfGrowHiX

これがセットされた場合，そのビューの右辺とオーナーの右辺との距離が一定に保たれます。

gfGrowHiY

これがセットされた場合，そのビューの下辺とオーナーの下辺との距離が一定に保たれます。

gfGrowAll

これがセットされた場合，そのビューは，大きさはそのまま同じにして，オーナーの右下隅に合わせて移動します。

gfGrowRel

これがセットされると，そのビューの大きさはオーナーの大きさに比例して変化します。このオプションは，デスクトップにリンクされる ***TWindow*** オブジェクト（もしくは ***TWindow*** の導出クラスのオブジェクト）にしか使用できません。この場合，ウィンドウは，ユーザがアプリケーションをノーマルモード（25行)/ハイレゾモード（31行）のどちらで動かしても，相対的な大きさが変わらないように働きます。このフラグは，ウィンドウ内のビューに対して使用するためのものではありません。

dragMode フラグバイト

dragMode データメンバは，ビューがドラッグされたときに，どのような振舞いをするかを決めます。***dragMode*** の各ビットの定義は以下のとおりです。

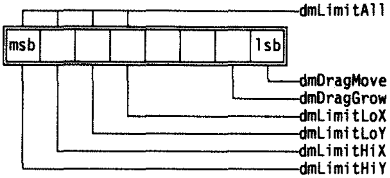


図 4.15
dragMode の各フラグビット

dmDragMove

このビットをセットした場合，ウィンドウの上フレームをクリックすることで，ドラッグを行なうことができます。

dmDragGrow

このビットをセットした場合，そのビューは拡大させることができます。

dmLimitLoX

これをセットすると，そのビューの左辺をオーナービューの外にもっていくことができなくなります。

dmLimitLoY

これをセットすると，そのビューの上辺をオーナービューの外にもっていくことができなくなります。

dmLimitHiX

これをセットすると，そのビューの右辺をオーナービューの外にもっていくことができなくなります。

dmLimitHiY

これをセットすると，そのビューの下辺をオーナービューの外にもっていくことができなくなります。

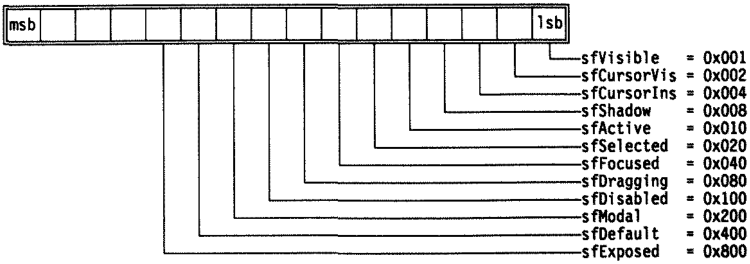
dmLimitAll

これをセットすると，そのビューのどの部分もオーナービューの外にもっていくことができなくなります。

state フラグと setState

この他，ビューは ***state*** というビットマップフラグを備えており，そのビューが表に見えているかかどうか，アクティブかどうか，ドラッグされているかどうか，といったビューのいろいろな状態をそのフラグに記録します。***state*** フラグビットの定義は**図 4.16** のとおりです。

図 4.16
state フラグのビットマッピング



各状態フラグの意味（ニーモニックからすぐわかるはず）については，第 16 章『**グローバルリファレンス**』の「**sfXXXX 状態フラグ定数**」で詳細を紹介しています。そこでこの節では，***state*** データメ

ンバの設定がどのようなメカニズムで行なわれているかについて説明します。

Turbo Vision がビューの ***state*** フラグを変更する場合には，そのビューの ***setState*** 仮想メンバ関数を使用します。ビューがフォーカスを獲得するか，フォーカスを放棄するか，あるいは選択されるかしたとき，Turbo Vision は ***setState*** を呼び出します。この点が，他のビットマップフラグの処理方法と異なるところです。他のビットマップフラグの場合には，通常，初期化時にセットされると，後で変更されることはありません（たとえば，ウィンドウがリサイズ可能とされているなら，**ずっと**リサイズ可能です)。これに対して，ビューの状態は，ビューが画面上にある間中，たびたび変更されます。このため Turbo Vision では，***setState*** を用意して，ビューの状態を変更するとともに，状態の変化に応じた処理を行なうことができるようにしています。

setState は，引数として状態(*aState*)と Boolean フラグ(*enable*)をとります。これで，その状態がセットされるかクリアされるかを示します。*enable* が *True* (ゼロ以外) なら，*aState* で示される各ビットが ***state*** でセットされます。*enable* が *False* (ゼロ) なら，対応する ***state*** のビットがクリアされます。ここまでは，基本的に，普通のビットマップフィールドの扱い方と同じです。異なる点は，ビューの状態を変更するときに，そのビューに何か**させたい**場合です。

状態の変化に伴う動作

ビューは，***setState*** が呼び出されると，その状態フラグの変化にしたがって，何らかの動作を起こすことがよくあります。たとえばボタンは，***state*** を監視し，フォーカスを得たら，色を黄色に変えます。以下は，***TView*** の下位オブジェクトにおける一般的な ***setState*** です。

```
void TButton::setState(ushort aState, Boolean enable)
{
  TView::setState(aState, enable);  // state ビットのセットまたはクリア
  if (aState & (sfSelected | sfActive))
      drawView();
  if (aState & sfFocused)
      makeDefault(enable);
};
```

TButton::setState の関数内にある ***TView::setState*** に注意してください。オーバーライドした ***setState*** メンバ関数では，必ず ***TView::setState*** を呼び出すようにしなければなりません。***TView::setState*** が，状態フラグの実際の設定（セット/クリア）を行ないます。その後で，そのときのビューの状態に基づく特別な処理を定義します。***TButton::setState*** の場合，それがアクティブなウィンドウであるかどうか，あるいは選択されているウィンドウであるかどうかを見て，自分自身を表示するかどうかを決めます。また，フォーカスを得ているかどうかを調べ，フォーカスを得ている場合には，その ***makeDefault*** メンバ関数を呼び出します。***makeDefault*** は，引数 *enable* にしたがって，そのフォーカスを摑んだままにしておくか手放すかします。

ある特定のビューの状態が変化したときに，ビューあるいはアプリケーションに変化を起こさせたい場合には，そのビューの ***setState*** をオーバーライドする必要があります。

状態に変化があった場合には，多くの場合，みなさんの書いたプログラムと Turbo Vision が協力して事を進めることになります。たとえば，テキストエディタで挿入モードがオンになっているときに，ブロックカーソルを表示させたいとします。

まず，エディタの挿入モードは，何らかのキー入力，たとえば ***INS*** キーで起動するようになっているでしょう。テキストエディタがフォーカスを得ているときに ***INS*** キーが押された場合には，テキストエディタが ***INS*** キーイベントを受け取ります。テキストエディタの ***handleEvent*** は，***INS*** イベントを受け取ると，***toggleInsMode*** メンバ関数を呼び出します。後のことは，Turbo Vision がやってくれます。***toggleInsMode*** は，以下に示すように，そのビュー（テキストエディタ）の ***setState*** を呼び出して，***sfCursorIns*** の状態を反転します。

```
void TEditor::toggleInsMode
{
    overwrite = Boolean(!overwrite);
    setState( sfCursorIns, Boolean(!getState(sfCursorIns)) );
}
```

getState メンバ関数は，引数で指定したコマンドが ***state*** で立っていたら *True* を返します。***state*** の ***sfCursorIns*** ビットは，カー

ソルの形状を決めます。挿入モードのときはブロックカーソルであり，上書きモードのときは通常のカーソルです。***state*** などのコマンドセットは，オーバーロード演算をいくつか定義している ***TCommandSet*** 型のオブジェクトを利用すれば，簡単に操作することができます。たとえば，

```
TCommandSet s, command;
...
s += command;
```

とすると，*command* というセットに *s* というセットを加えることになります。同様に，－＝は，コマンドセットからコマンドを除外します。コマンドを**有効にしたり無効にしたりする**処理を記述することもできます。また＝＝と！＝で，２つのコマンドセットが等しいかどうかを検査することもできます。

ビューに何色を使うか？

コンピュータの画面に何色を使うのが一番いいのかについては，決まった見解はないようです。それで，Turbo Vision では，画面に表示させるビューの色を変えることができるようにしています。Turbo Vision では，カラーパレットを扱うための ***TPalette*** クラスを使って，これを実現しています。

カラーパレット

標準的なビュー全部のパレットが，第 13 章『オブジェクトリファレンス』で紹介されています。

Turbo Vision において，ビューが自分自身を表示させる場合，特定の色を指定するのではなく，そのビューのパレット内の何番目の色という指定をして，表示を行なわせます。たとえば，***TScroller*** のパレットは，以下のようなものです。

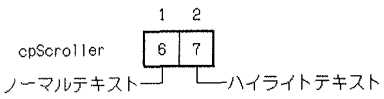
```
#define cpScroller "¥x06¥x07"
```

カラーパレットは，***TPalette*** オブジェクトです。メンバ関数が内部的なことはすべて面倒をみてくれるので，通常，パレットの内部的な構造について関知する必要はありません。***cpScroller*** は，パレットエントリを表わす 2 個の文字からなる文字列であるとみなすことができます。***TScroller*** パレットのレイアウトは，次のようになります。

```
// パレットのレイアウト
// 1 = ノーマル
// 2 = ハイライト
```

しかし，これは次のような見方をしたほうが使いやすくなります。

図 4.17
TScroller のデフォルトのカラーパレット



getColor は，TView のメンバ関数です。

これは，スクロールオブジェクトが 2 種類のテキスト，すなわちノーマルとハイライトの 2 種類を表示できるということを意味して

います。各ビューのデフォルトの色は，パレットのエントリで決まります。通常のテキストを表示する場合には，***draw*** メンバ関数は ***getColor***(1)を呼び出すようにします。つまり，パレットの1番目の色を使うということです。テキストをハイライトさせたい場合には，***getColor***(2)を呼び出します。

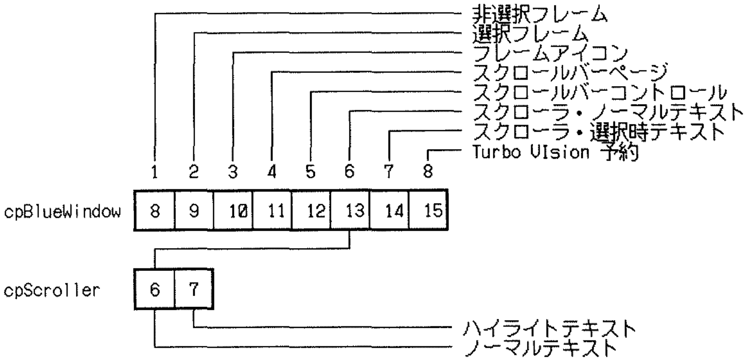
デフォルト以外の色の選択については，次の節で説明します。

デフォルトの色を使って表示する場合には，これだけ知っていれば充分です。オブジェクトの組み合わせが普通のものであるぶんには，体裁のよい配色となるように，パレットは設定してあります。

カラーパレットの内部構造

パレットのエントリは，色そのものではなく，オーナーの所有するパレットに対するインデックスとなっています。スクロールオブジェクトをウィンドウに挿入した場合，通常のテキストにはスクロールオブジェクトのパレットにおけるノーマルテキストカラーを使用します。そのノーマルテキストカラーは6という数が入っています。それを実際の色に直すには，そのオーナーのパレットの6番目のエントリを引っぱってきます。図4.18には，***TWindow*** のパレットを示します。

図4.18
ウィンドウに対するスクロールオブジェクトのパレットのマッピング



TWindow のパレットの6番目のエントリは 13であり，これはそのウィンドウのオーナー（デスクトップ）のパレットへのインデックスになっており，さらにそれがデスクトップのオーナーであるアプリケーションのパレットに対するインデックスとなります。

TDeskTop のパレットは，ヌルパレットになっています。ヌルパレットというのは，何も変換を行なわないパレットのことで，１番目のエントリには1が，２番目のエントリには2が入っているという，“平行な”あるいは“透明な”パレットであると見ることができます。

アプリケーションももちろんパレットを持ちますが，それはTurbo Visionのアプリケーションに挿入されうるすべての要素に対する64個のエントリを備えた大きなパレットです。アプリケーションパレットの13番目のエントリは，0x1Eとなっています。アプリケーションで行き止まりですから(アプリケーションにはオーナーがない)，パレットのマッピングはそこで終わります。

それぞれの数字の表わす意味については，第13章の「TProgram」のパレットの解説をご覧ください。

パレットを色として捉えないでください。パレットは，表示内容の類に入ります。

さて0x1Eについてですが，これは背景色が1，文字色が0xE(14)のテキスト属性のことを意味しており，これによって青の背景に黄色の文字が表示されることになります（拡張アトリビュートモードの場合)。改めて言いますが，これを青に黄色という関係で捉えないでください。ウィンドウテキスト用の通常のテキスト色，つまり13番目のエントリでテキストを表示するという捉え方をしてください（拡張アトリビュートを使用しているときは13番目のエントリは0x1Eとなり，青の背景に黄色で文字が表示されます。ノーマルアトリビュートでの13番目のエントリでは，0xE1となり黒地の白い文字が表示されます。拡張アトリビュートとノーマルアトリビュートでは，最終的な数値の意味が違っています)。

getColor メンバ関数

カラーパレットのマッピングは，仮想メンバ関数 ***TView::getColor*** によって行なわれます。***getColor*** は，表示しようとしているオブジェクトから，そのオーナーへ，さらにそのオーナーのオーナーへ，という具合いにビューツリーをさかのぼっていき，アプリケーションオブジェクトのところに行き着きます。このチェーン上の各オブジェクトについて，***getColor*** が各オブジェクトの ***getPalette*** を呼び出します。最終的な結果として，カラー属性を得ます。

ビューのパレットには，そのオーナーのパレットに対するオフセ

ットが格納されています。ただし，アプリケーションのパレットの場合には，オフセットではなく，カラー属性が入っています。

デフォルトカラーのオーバーライド

通常アトリビュートモードでは，黒い背景に水色の表示がデフォルトです。

当然ながら，色を変えるにはパレットを変えます。スクローラの通常のテキスト色が気に入らないというのであれば，まず第一に，スクロールオブジェクトのパレットのエントリ 1（通常のテキスト色のエントリ）を，おそらく 6 から 5 に変えようと考えることでしょう。そうすると，スクロールオブジェクトの通常のテキストは，ウィンドウのパレットのスクロールバーコントロールのエントリにマッピングされることになります（デフォルトでは，背景が水色，文字が青）。**5 というのが色でないこと**を忘れないでください！ ここで行なったのは，通常のテキストをスクロールバーと同じ配色にしてほしいとスクローラに伝えただけです！

では，この色が気に入らなければ，どうすればよいでしょうか。***TApplication*** のパレットのノーマルウィンドウテキストのエントリを変えればよいのです。***TApplication*** のパレットはヌルパレットではありませんので，アプリケーションのパレットが，ウィンドウ内のすべてのビューの配色を決定します。これを自分のカラーマントラにします。色は絶対的な指定をするのではなく，オーナーのパレットによって決定されます。

こうするのには訳があります。つまり，ウィンドウはどれも同じ配色にしたいであろうということです。ウィンドウごとにいちいちどの色を使うかを指定したくはないはずです。後で気が変わったら（あるいは，ユーザが色をカスタマイズできるようにした場合には），ウィンドウ**ごとに**エントリを変えていくことになります。

また，スクローラやその他のインテリアも，最初に意図したものとは別のウィンドウに挿入された場合でも，配色のことで煩わされる必要はありません。たとえば，スクローラをウィンドウではなくダイアログボックスに挿入した場合，同じ色では表示されず（デフォルトの場合），ダイアログボックスの通常のテキスト色で表示されます。

ビューのパレットを変えるには，仮想メンバ関数 ***TView::***

getPalette をオーバーライドします。通常のテキスト色ではなく，ウィンドウフレームの色を使って自分自身を表示するスクローラクラスを新たに作るためには，クラスの宣言および定義は，以下のようなものとなります。

```
class TMyScroller : public TScroller
{
    ...
    virtual TPalette& getPalette() const;
    ...
};

TPalette& TMyScoller::getPalette() const
{
    static TPalette palette( cpFrame, sizeof( cpFrame ) - 1 );
    return palette;
}
```

この ***TPalette*** コンストラクタは，引数にパレット文字列とその長さ（1バイト目の長さバイトの分は差し引く）をとり，新しいパレットオブジェクトを返します。各ビュークラスにある ***getPalette*** はデフォルトで，それぞれのクラスに割り当てられている標準パレットオブジェクトを返すようになっています。

新たに色を追加するには

ウィンドウクラスに別の色を追加して，新たに作り出すビューに豊富な配色をほどこしたい場合があります。たとえばスクローラに第3番目の色を持たせ，統合環境のエディタがブレークポイントに使っているような，違ったハイライトのさせ方をしたいとします。このためには，現在の ***TWindow*** から新たなクラスを導出して，デフォルトのパレットを追加します。以下がその例です。

```
#define cpNewPalette cpBlueWindow "¥x54"

class TMyWindow : public TWindow
{
    ...
    virtual TPalette& getPalette() const;
    ...
}
```

```
TPalette& TMyWindow::getPalette() const
{
    static TPalette palette (cpNewPalette, sizeof( cpNewPalette) - 1);
    return palette;
}
```

これで，*TMyWindow* のパレットには，ハイライトをこれまでとは違うものにする新しいエントリが格納されたことになります。***cpBlueWindow*** は，***TWindow*** のデフォルトパレットを表わす文字列定数です。

```
#define cpBlueWindow "¥x08¥x09¥x0A¥x0B¥x0C¥x0D¥x0E¥x0F"
```

新しいウィンドウパレットを利用するためには，以下のように，*MyScroll::getPalette* を変更する必要があります。

```
TPalette& TMyScroller::getPalette() const
{
    const char *cpMyScroller = "¥x06¥x07¥x09";
    static TPalette palette( cpMyScroller, sizeof(cpMyScroller)-1 );
    return palette;
}
```

これで，スクローラのパレットエントリ3番は，新しいハイライトカラーを指定するようになったわけです（この場合は，赤地に白（拡張アトリビュートモードの場合））。*cpMyScroller* を使って，オーナーのパレットの9番目にアクセスする新しい ***getPalette*** を利用する場合には，オーナーが *cpMyWindow* パレットを使用していることを確認してください。エントリが8個しかないパレットに対して，9番目をアクセスした場合には，結果は不定のものとなります。

第5章
イベント駆動型プログラミング

Turbo Vision の目的は，アプリケーション開発用のちゃんとした枠組みを提供することにより，開発者がアプリケーションの“肉”の部分だけに集中できるようにすることです。Turbo Vision のツールとしての役割は，ウィンドウの組み込みをサポートすることと，イベントを処理することの2点です。ビューについては，第4章で説明しました。そこでこの本章では，プログラムを作成する上でのイベント回りのことについて説明することにします。

Turbo Vision に魂を入れる

Turbo Vision のアプリケーションがイベント駆動型であるということについては，すでに説明しました。また，イベントとは，アプリケーションが対処すべき何かが発生したということであると，簡単に定義しました。

ユーザ入力の読み込み

従来の 手続き型プログラムでは，キーボードやマウスなどからユーザ入力を読み込むコードをループさせるのが一般的です。そしてループ内で，入力に応じた処理内容を決定します。たとえば，以下のように，関数を呼び出したり，ループさせるためにどこかに分岐させて，またユーザ入力の読み込みを行なうというわけです。

```
quit = False;
do
{
  B = ReadKey();
  switch( B )
  {
    case 'i':
    case 'I':
        invertArray();
        break;
    case 'e':
    case 'E':
        editArray();
        break;
    case 'g':
    case 'G':
        graphicDisplay();
        break;
    case 'q':
    case 'Q':
        quit = True;
        break;
    default:
        break;
  }

} while (!quit);
```

イベント駆動型プログラムの構造は，これとまったく異なるというわけではありません。実際，このような処理を行なわない対話型プログラムを考えることは難しいでしょう。しかしながら，プログラミング上は，イベント駆動型プログラムは違ったふうに見えます。

Turbo Vision のアプリケーションでは，プログラマがユーザ入力の読み出しを考える必要はありません。Turbo Vision がその面倒をみてくれるからです。Turbo Vision は，入力を**イベント**（**struct *TEvent*** 型のオブジェクト）に仕立てて，プログラム内のしかるべきビューにそのイベントを発送します。つまり，ユーザプログラムでは，関係する入力に対する処理方法だけが書かれてあるだけでよく，入力ストリームの選別を行なって処理の対象を探し出すという作業は必要ありません。

たとえば，ユーザが休止状態のウィンドウをクリックした場合，Turbo Vision がマウスの動作を読み出し，それをイベントオブジェクトに仕立てて，その休止状態のウィンドウにイベントオブジェクトを送ります。

従来のプログラミング方法に慣れている人なら，ここまで説明すれば，次のように考えることでしょう。“わかった。つまり，これからはユーザ入力を読み出す必要はないのだ。そのかわりにやるべきことは，マウスクリックイベントの読み出し方と，休止状態のウィンドウをアクティブにするためにそのウィンドウにどのようなことを伝えればよいのかを理解すればよいわけだ。”しかし実際には，そのようなコードすら記述する必要はないのです。

ビューは，たいていのユーザ入力は，自分自身で処理することができます。ウィンドウは，オープン，クローズ，移動，選択のされ方，リサイズなどを自分で処理できます。メニューも，オープン，ユーザとの対話，クローズを自分で処理します。またボタンも，プッシュされたときの処理，ボタン同士の相互作用，色の変更を自分で行ないます。スクロールバーも，ユーザの操作に対してどのような処理を行なうかを知っています。休止状態のウィンドウも，プログラムで何も指定しなくても，自分自身でアクティブになることができます。

それでは，プログラマの仕事は何なのでしょうか。それは，1つには，新たな機能を備えたビューを新たに定義することです。そのビューは，プログラマが定義した何らかのイベントを処理できる必

要があります。また，そのビューは，標準的なコマンドを受けたとき処理方法や，自分自身のコマンド（“メッセージ”）を他のビューに発行する方法を心得ておく必要があります。そのメカニズム自体はすでに決まっています。プログラマが行なうことは，コマンドを発生させることと，ビューがコマンドを受けたときにどんな処理をビューにさせるかを指定するということです。

ではいったいイベントとは，プログラムから見れば，実際どのようなものなのでしょうか。また Turbo Vision は，プログラマのかわりにどのようなイベント処理を行なってくれるのでしょうか。

イベントの性質

イベントとは，アプリケーションが対処すべき個々の出来事についての情報が書かれた小さな小包であると考えるのが一番わかりやすいでしょう。1つ1つのキー入力やマウスの動き，あるいはプログラムのどこか別の部分によってもたらされた何らかの状況が，それぞれイベントとなります。イベントは，これ以上小さく分解することはできません。したがって，ユーザが1単語をタイプしたとすると，それは1個のイベントではなく，1個1個のキー入力のイベントを連ねたものとなります。

すべての ***TEvent*** オブジェクトの中心に，***what*** と呼ばれる ***ushort*** 型のデータメンバがあります。***what*** に格納される数値が，発生したイベントの種類を表わし，イベント構造体の残りの部分は，各イベントに固有な情報を表わします。たとえば，キー入力イベントの場合には，キーボードスキャンコードが，マウスイベントの場合には，マウスの位置およびマウスボタンの状態についての情報が格納されるといった具合いです。

イベントの種類によって，送り先のオブジェクトに送られるルートが異なりますので，まずは Turbo Vision が認識するイベントの種類について見ておくことにしましょう。

イベントの種類

ここで，***TEvent::what*** がとり得る値について少し詳しく見ておきます。イベントには，基本的に4つの種類があります。マウスイベント，キーボードイベント，メッセージイベントおよび“なし”イベントです。それぞれの種類でマスクが定義されており，オブジェクトからは，発生したイベントがどの種のものであるかをすぐに判断することができるようになっており，そのイベントが何であったかを特定しなくてもよいようになっています。たとえば，マウスイベントには4種類の個別的なイベントがありますが，そのどれであるかを調べなくても，イベントフラグがそのマスクに合致するか

どうかだけを調べればよいわけです。

```
TEvent event;
 ...
if (event.what & (evMouseDown | evMouseUp | evMouseMove |
                  evMouseAuto)) != 0 { ... }
```

とするかわりに,

```
if (event.what & evMouse) != 0  { ...
```

とすればよいわけです。

イベントの種類を判断するために使用されるマスクには,***evNothing*** (“なし”イベント用), マウスイベント用の ***evMouse***, キーボードイベント用の ***evKeyboard***, およびメッセージ用の ***evMessage*** があります。

図 5.1 は, イベントマスクビットの定義です。

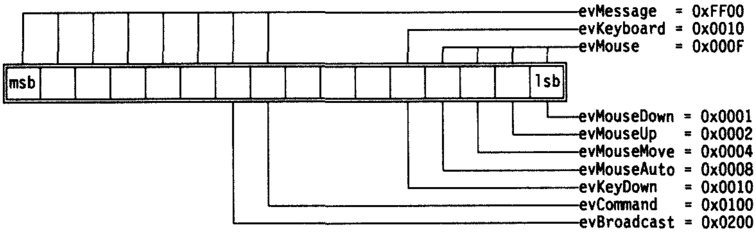


図 5.1
TEvent.what データメンバのビットマップ

マウスイベント

マウスイベントには, 基本的に 4 種類のイベントがあります。ボタンごとのクリックアップおよびクリックダウン, マウスの位置の変化, および“オート”の各イベントです。マウスボタンを押すと, ***evMouseDown*** イベントが発生し, ボタンを離すと, ***evMouseUp*** イベントが発生します。マウスを動かすと, ***evMouseMove*** イベントが発生します。また, ボタンを押したままにしておくと, Turbo Vision は, ある一定の時間間隔で ***evMouseAuto*** イベントを発生し, アプリケーションが連続的なスクロールなどの動作を行なえるようにします。すべてのマウスイベントオブジェクトにマウスの位置が記録され, それによって, そのイベントを処理するオブジェクトは, イベント発生時にマウスがどこにあったかを知ることができます。

キーボードイベント

キーボードイベントの場合は，もっと簡単です。キーを押すと，Turbo Vision が ***evKeyDown*** イベントを発生します。そのイベントには，どのキーが押されたかが記録されます。

メッセージイベント

メッセージイベントには，3種類の味付けが施されます。コマンド，ブロードキャスト，ユーザメッセージの3種類です。この3つは，処理のされ方によって区別されるのですが，これについては後で説明します。基本的に，コマンドの場合，***what*** データメンバに ***evCommand*** のフラグが立ち，ブロードキャストの場合には ***evBroadcast*** が立ちます。ユーザ定義メッセージの場合には，ユーザが定義した何らかの定数がフラグとして立ちます。

“なし”イベント

“なし”イベントは，実際には無効なイベントです。これは，処理済みとなり，イベントとしての役割を終えたものです。イベントオブジェクトの ***what*** データメンバに ***evNothing*** の値が入っている場合，そのイベントオブジェクトには処理に必要な情報は含まれていません。

Turbo Vision のオブジェクトがイベントの処理を終了すると，***clearEvent*** というメンバ関数を呼び出します。***clearEvent*** は，そのイベントが処理済みであるという意味で，***what*** データメンバに ***evNothing*** を書き戻します。オブジェクトが ***evNothing*** イベントを得た場合，すでに別のオブジェクトによって処理済みのイベントなので，単に無視すればよいわけです。

イベントとコマンド

たいていのイベントは，最終的には，何らかのコマンドに変換さ

れることになります。たとえば，ステータスラインの項目上でマウスをクリックすると，マウスイベントが発生しますが，そのイベントがステータスラインオブジェクトに送られると，そのオブジェクトはマウスイベントに対応させてコマンドイベントを発生します。その際，***command*** データメンバには，ステータスライン項目にリンクされたコマンドによって指定された値が入れられます。たとえば，GRPH-X Exit をマウスクリックすると，***cmQuit*** コマンドが発生しますが，アプリケーションは，そのコマンドをシャットダウンし終了せよという命令として解釈します。

イベントの運ばれる経路

Turbo Vision のビューは，“語りかけられたときだけ語れ”という原則にしたがって動作します。つまりビューは，自発的に入力を探しにいくのではなく，そのビューによって処理されるべきイベントが発生したことをイベントマネージャが伝えてくるまで受身的に待つわけです。

Turbo Vision のプログラムに期待どおりの動作をさせるには，イベントが発生したときにビューに何をさせるかを指定すればよいというだけではなく，イベントがそのビューにどのように運ばれてくるかについても理解しておく必要があります。イベントをしかるべき場所に送るための鍵となるのが，イベントを正しい**ルートで運ぶ**ということです。イベントによっては，アプリケーション全体にブロードキャストされるものもありますが，それ以外のイベントは，送り先がプログラム内の特定の部位に限定されます。

イベントの出所は？

第１章『**車輪の継承**』で述べたように，***TApplication*** で処理の中心としてループを行なっている ***run*** メンバ関数は，***TGroup.execute*** を呼び出します。***TGroup.execute*** は，基本的に以下のようなループを構成しています。

```
TEvent event;
event.what = evNothing;    // 何もイベントがないことを示す
do {
   if (event.what != evNothing) eventError(event);
   getEvent(event);          // イベントを得て
   handleEvent(event);       // 適当な箇所にそのイベントを送出する
} while (endState == continue); // 終了フラグが立つまで
```

getEvent, handleEvent, eventError については，それぞれ，194, 187, 191 ページで詳しく説明しています。

getEvent は，基本的な機能として，イベントとなるべきことが何か起こっていないかどうかを調べて回ります。何か起こっていれば，それに応じた値をイベントオブジェクトにセットします。そして ***handleEvent*** が，そのイベントをしかるべきビューに発送します。イベントがこのループに戻されてくるまでの間に処理され（クリアされ）なかった場合には，廃棄イベントとするために ***eventError*** が呼び出されますが，***eventError*** は，何も指定しなければ（デフォルトでは）何も行ないません。

イベントの送り先はどこか？

イベントは，**常に**，まず現在のモードつきビューに送られます。通常に動作している場合，一般にアプリケーションオブジェクトがそのモードつきビューとなります。モードつきダイアログボックスを実行している場合には，そのダイアログボックスオブジェクトが当該のモードつきビューとなります。いずれにせよ，モードつきビューがイベント処理を開始します。イベントがそこから先どこに行くかは，イベントの内容によります。

イベントがどの種のものであるかによって，イベントの送られるルートは3通りに分かれます。その3通りとは，ポジショナルイベント，フォーカスイベント，およびブロードキャストイベントです。以下で，各種のイベントがどのようなルートで発送されるかについて説明します。

ポジショナルイベント

ポジショナルイベントは，実際上は，常にマウスイベント（***evMouse***）です。

Z オーダーについては，第4章の135ページで説明してあります。

モードつきビューは，まずポジショナルイベントを受け取り，イベントの発生した位置にあるサブビューを見つけ出すまで，Z オーダーにしたがってサブビューの探索していきます。見つかったらイベントをそのビューに渡します。ビューはオーバーラップすることができるので，その位置に複数個のビューがある場合があります。

Zオーダーにしたがって探索を進めることによって，その位置にある最前面のビューがイベントを受け取ることになります。結局それが，ユーザのクリックしたビューだったということです！

オブジェクトがターミナルビュー（サブビューを持たない）であるか，あるいはイベントの発生した位置にサブビューがない（たとえば，ダイアログボックス上の何もないところでクリックした場合など）ために，そのオブジェクトがイベントを渡すべきビューを見つけることができなくなるまで，上のプロセスは続けられます。その時点で，ポジショナルイベントの発生したオブジェクトのところまで，そのイベントが届けられたわけで，そのオブジェクトがイベントの処理を行ないます。

フォーカスイベント

フォーカスビュー，およびフォーカスチェインについては，第4章の144ページを参照してください。

フォーカスイベントは，一般に，キー入力（***evKeyDown***）かコマンド（***evCommand***）であり，フォーカスのチェインに沿って，どんどん渡されていきます。

まず最初に，現在のモードつきビューが，フォーカスイベントを受け取り，そのビューの管理下にある選択されているサブビューにそのイベントを渡します。そのサブビューが，さらに，選択されているサブビューを所有している場合には，イベントをそのサブビューに渡します。このプロセスは，ターミナルビューにビューに達するまで続けられます。結局，そのターミナルビューがフォーカスビューということになります。そのフォーカスビューが，フォーカスイベントを受け取り，処理を行ないます。

フォーカスビューでないビューがフォーカスイベントを処理する場合もあります。これについては，179ページを参照してください。

フォーカスビューが，受け取ったイベントを処理できない場合には，そのイベントをフォーカスチェインに沿ってオーナーに戻します。このプロセスは，イベントが処理されるか，あるいはそのイベントがモードつきビューのところまで戻ってくるまで繰り返されます。イベントがモードつきビューのところまで戻されたとしても，そのモードつきビューがイベントの処理を行なえない場合には，モードつきビューは ***eventError*** を呼び出します。こうなった場合には，**廃棄イベント**となります。廃棄イベントについては，191ページを参照してください。

キーボードイベントが，フォーカスイベントを非常に明瞭に説明

してくれる例です。たとえば，Borland IDE（総合開発環境）のデスクトップ上で，いくつかのエディタウィンドウにファイルをオープンしていたとします。何かキーを入力する場合，ユーザは，ある特定のファイルにその文字を受け取らせたいわけです。入力した文字を実際にそのファイルに届けるために，Turbo Vision がどのようなことを行なうかについて，以下で見てみることにしましょう。

ユーザがキー入力を行なうと，***evKeyDown*** イベントが発生し，現在のモードつきビューである ***TApplication*** オブジェクトに送られます。***TApplication*** は，その管理下にあって選択されているビュー，すなわちデスクトップにイベントを渡します（デスクトップは，常に ***TApplication*** 下で選択されているビューです）。さらにデスクトップは，その管理下にあって選択されているビュー，すなわちアクティブなウィンドウ（フレームが太線になっているウィンドウ）に，そのイベントを渡してやります。そのエディタウィンドウも，フレーム，スクロールするインテリアビュー，および2本のスクロールバーといったサブビューを備えています。このうち，インテリアだけが選択可能なビューなので（したがって，デフォルトで選択されている），キーボードイベントは，そのインテリアに渡ります。そのインテリアビュー，すなわちエディタは，サブビューを持たないので，それが ***evKeyDown*** イベント内の文字を処理するかどうかを決めることになります。

ブロードキャストイベント

ブロードキャストイベントは，一般に，ブロードキャスト（***evBroadcast***）であるか，ユーザ定義のメッセージであるかのいずれかです。ブロードキャストイベントの場合，ポジショナルイベントあるいはフォーカスイベントと同じような経路はたどりません。ブロードキャストについては，送り先が定義されておらず，現在のモードつきビューの管理下にある**すべての**サブビューに送られます。

現在のモードつきビューがイベントを受け取ると，Zオーダーにしたがって，その各ビューにイベントを渡していきます。そのサブビューがグループであった場合には，さらにそのサブビューにイベントを渡していきます。このプロセスは，モードつきビューの管理下にあるすべてのビューが（直接的に，あるいは間接的に）そのイ

ベントを受け取るまで続けられます。

ブロードキャストは，message関数を使用することで，特定のオブジェクトに渡すこともできます。

ブロードキャストイベントは，通常，ビュー同士の通信に使用されるものです。たとえば，ファイルビュアーのスクロールバーをクリックした場合，スクロールバーは，テキストビューに対して，テキストをずらして表示するよう知らせる必要があります。このためには，“変更あり！”というビューをブロードキャストすることで，テキストなどの他のビューがそれを受け取り，それに応じた動作を行なえるようにするわけです。詳細については，この章の196ページを参照してください。

ユーザ定義のイベント

Turbo Visionやイベントの使い勝手がわかるようになると，イベントレコードの ***what*** データメンバの上位ビットを使って，まったく新しい種類のイベントを定義したくなるのではないでしょうか。デフォルトとした場合，Turbo Visionは，そのようなイベントをすべてブロードキャスイベントとして送出します。しかし，新たに定義したイベントを，フォーカスあるいはポジションにしたい場合もあるでしょう。Turbo Visionは，そのためのメカニズムも備えています。

マスク内のビットを扱う方法については，第10章『ヒントとコツ』で説明します。

Turbo Visionでは，***positionalEvents*** と ***focusedEvents*** の2つのマスクを定義しています。それぞれ，送出されるイベントが位置を基準とするかフォーカスを基準とするかに応じて，イベントレコードの ***what*** データメンバ内のビットが定義されています。デフォルトとしては，***positionalEvents*** には ***evMouse*** の全ビットが含まれ，***focusedEvents*** には(***evKeyboard*** | ***evCommand***)が含まれます。どこか別のビットを新しいイベントとして定義し，位置かフォーカスのいずれかを基準として送出するようにしたい場合は，単にマスクにそのビットを加えるだけで，そうすることができます。

イベントのマスク

すべてのビューオブジェクトが ***eventMask*** と呼ばれるビットマップ化された ***ushort*** 型のデータメンバを備えており，そのビュー

が処理するイベントは，それによって指定されます。***eventMask*** の各ビットは，***TEvent::what*** フィールドのそれに対応しています。あるイベントに対するビットが立っていれば(＝1)，ビューはそのイベントを処理するということです。逆に，そのビットがクリアされていれば（＝0），ビューはそのイベントを無視します。

フェーズ

場合によっては，フォーカスビュー以外のビューにフォーカスイベントを処理させたい場合があります(とくにキー入力の場合)。たとえば，スクロールテキストウィンドウを開いているときに，キー入力によってテキストをスクロールしたい場合がありますが，そのテキストウィンドウがフォーカスビューですので，キー入力イベントは，ビューをスクロールするスクロールバーに送られずに，テキストウィンドウに送られます。

Turbo Vision には，フォーカスビュー以外のビューもフォーカスイベントを処理できるようなメカニズムが備わっています。この章の「フォーカスイベント」で，イベントのルートについて説明したことは，一般的にはそのとおりなのですが，フォーカスチェインの厳密なルートについては例外が2つあります。

モードつきビューが処理すべきフォーカスイベントを得た場合，実際には，ルーティングに3つの“フェーズ”があります。

- ■イベントは，どれか ***ofPreProcess*** オプションフラグの立っているサブビューに送られる（送られる順番はZオーダーにしたがう)。
- ■もしそのイベントが，どのサブビューからもクリアされないなら，フォーカスビューに送られる。
- ■それでもそのイベントがクリアされないなら，イベントは，どれか ***ofPostProcess*** オプションフラグの立っているサブビューに送られる（この場合も順番はZオーダーにしたがう)。

したがって先の例では，フォーカスの当たっているテキストビューに向けられたキー入力を，スクロールバーに得させたいのであれば，そのスクロールバーを初期化する際に ***ofPreProcess*** オプショ

ンフラグをセットしておく必要があります。例題プログラムTVGUID09.CPPを見ると，インテリアビューのスクロールバーは，どれも ***ofPostProcess*** ビットがセットされているのがわかります。このプログラムを書き換えて，***ofPostProcess*** ビットをセット**しない**ようにすると，キー入力によるスクロールは無効にされます。

またこの例の場合は，***ofPreProcess*** を立てても ***ofPostProcess*** を立てても大した違いはありません。どちらでもちゃんと働きます。この例では，フォーカスビューがキー入力イベントを処理しないので（***TScroller*** 自体は，キー入力に関しては何も行なわない），そのイベントがスクローラに送られる**前であろうが後であろうが**，スクロールバーはそのイベントを獲得することができるわけです。

しかしながら，このような場合には，一般に，***ofPostProcess*** を使用したほうがよいでしょう。というのは，そのほうが柔軟性に富むからです。たとえば，後で，キー入力を必要とする機能をインテリアに追加したいという場合，そのイベントは，フォーカスビュー（***ofPreProcess*** の立っているもの）に渡される前に，スクロールバーに取られてしまうことになれば，インテリアはそのイベントを絶対に処理できなくなることになります。

☞ フォーカスビューよりも先にフォーカスイベントを獲得する**必要がある**場合でも，オプションを立てるのは必要最小限にとどめておいたほうがよいでしょう。そうすることによって，将来（誰か他の人も）そのオブジェクトから新たに何かを導出できることになります。

フェーズデータメンバ

すべてのグループが以下の enum ***PhaseType*** 型のフラグをデータメンバとして備えています。

```
enum phaseType { phFocused, phPreProcess, phPostProcess}
```

オーナーの ***phase*** フラグを調べることによって，ビューは，現在処理中のイベントが，フォーカスビューへの送出前のものなのか，送出後のものなのか，あるいはフォーカスビューによる処理中のものなのかを知ることができます。ビューによっては，フェーズにしたがって，別のイベントを探したり，同じイベントに対して異なる

方法で処理することがあり，そのような場合には，*phase* フラグを調べる必要があります。

たとえば，入力行が1つと“All right”というラベルをもつボタンを1個備えた簡単なダイアログボックスの場合を考えてみましょう。そのボタンのホットキーは ***A*** だとします。通常のダイアログボックスのコントロールであれば，フェーズに関知する必要はありません。コントロールには，たいていデフォルトで ***ofPostProcess*** がセットされており，したがって，タイプされた文字がホットキーであれば，そのキー入力（フォーカスイベント）は，コントロールに届けられ，そのコントロールがフォーカスを獲得することができます。たとえば，***A*** を押せば，フォーカスは，“All right”ボタンのところに移ります。

しかし，入力行がフォーカスを獲得していたとすると，キー入力は，その入力行によって処理され挿入されることになります。したがって，***A*** のキーを押した場合，フォーカスビューがそのイベントを処理することになるので，入力行に“A”が入り，ボタンはそのイベントを決して獲得できないことになります。そこで，***A*** キーが処理される前に，ボタンに調べさせればよいと考えるかもしれません。そうすれば，フォーカスビューがそれを処理する前にホットキーを処理することができるのではないかと。しかし残念ながら，こうすると，入力行に“A”という文字をタイプすることができなくなります。

これは，実際には，簡単に解決することができます。フォーカスビューがイベントを処理する前後で，ボタンに他のホットキーを調べさせればよいのです。とくにデフォルトにした場合には，ボタンは，フォーカスビューによる処理前には **GRPH-文字**の形式でホットキーかどうかを判断し，処理後は，通常の文字のみの形式で判断します。ダイアログボックスでいつでも **GRPH-文字**のホットキーを使えるのは，このためです。ただし，通常の文字を使用できるのは，フォーカスコントロールがキー入力をすぐにもっていかない場合に限ります。

これは，次のようにして簡単に実現できます。デフォルトでは，ボタンは，***ofPreProcess*** フラグも ***ofPostProcess*** フラグも立てています。したがって，フォーカスビューがフォーカスイベントを調べる前**と**後で，それを調べることができます。ただし，フォーカス

コントロールがすでにそのイベントを扱っていた場合には，***handleEvent*** 内ではキー入力を検査することしか行ないません。

```
switch (event)
{
    ...
    case evKeyDown:
    {
       c = hotKey(title);
       if ( (event.keyCode == getAltCode(c)) ||
          (owner->phase == phPostProcess) && (c != '0') &&
          (toupper(event.charCode) == c) ||
          ((state & sfFocused) != 0) &&
          (event.charCode == ' ') )
          {
             pressButton();
             clearEvent(event);
          }
     }
     break;
     ...
}
```

コマンド

ほとんどのポジショナルイベントおよびフォーカスイベントが，それを処理するオブジェクトによってコマンドに変換されることになります。つまり，オブジェクトは，マウスクリックやキー入力が発生したら，たいていはコマンドイベントを発行するわけです。

たとえば，ステータスラインをクリックすると，ポジショナル(マウス)イベントが発生しますが，アプリケーションは，クリックがステータスラインの管理領域下のものであるとして，そのイベントをステータスラインオブジェクトである ***statusLine** に渡します。

statusLine は，クリックのあった領域がどのステータス項目によって管理されているかを判断し，その項目のステータス項目レコードを読み出します。項目には，通常，コマンドがリンクされていますので，***statusLine*** は，それを基にペンディングイベントオブジェクトを作成します。***what*** データメンバには ***evCommand*** をセットし，***command*** データメンバにはそのステータス項目にリンクされていたコマンドをセットします。その後で，マウスイベントをクリアします。これで，***getEvent*** が次に扱うイベントは，いま作成したコマンドイベントとなります。

コマンドの定義

Turbo Vision には，定義済みのコマンドがたくさんあります。さらにその他に，新たにコマンドをたくさん定義することもできます。ビューを新たに作成する場合には，そのビューを呼び起こすためのコマンドも作成することになります。コマンドには，どのような名前を付けてもかまいませんが，Turbo Vision では，慣例として，コマンド名を **cm** で始めるようにしています。コマンド作成といっても，実際は単純で，定数を作成するだけです。

```
const cmConfuseTheCat = 100;
```

Turbo Visionは，0〜99までのコマンド，および256〜999のコマンドを内部的なコマンドとして予約しています。アプリケーション用のコマンドとしては，100〜255および1000〜65,535が開放されています。

コマンドを2つのグループに分けているのは，0〜255のコマンドだけを無効にすることができるようにしているからです。Turbo Visionのコマンドには，無効にすることのできるものと，標準コマンドおよび内部作業用として無効にすることのできないものとがあります。その他のコマンドについては，アプリケーションで任意に定義することができます。

コマンドの範囲による分類を，**表5.1**にまとめておきます。

表5.1
Turbo Visionのコマンドの範囲

範囲	**予約**	**無効**
0〜99	予約済み	可
100〜255	空き	可
256〜999	予約済み	不可
1,000〜65,535	空き	不可

コマンドのリンク

メニュー項目やステータスライン項目を作成したときには，それにコマンドをリンクします。ユーザがその項目を選択すると，イベントレコードが生成されますが，その ***what*** データメンバには ***evCommand*** が，***command*** データメンバには，リンクされたコマンドの値が設定されます。コマンドは，Turbo Visionの標準コマンドにすることも，自分で定義したコマンドにすることもできます。同時に，メニュー項目やステータスライン項目に自分のコマンドをリンクします。ホットキーにコマンドをリンクすることもできます。そうすれば，メニューやマウスを使うかわりに，ショートカットとしてキーを1個押すことで，コマンドを呼び起こすことができます。

☞ 銘記しておいていただきたいことは，コマンドを定義したとしても，イベントレコードにそのコマンドがセットしてあったときにどんな動作が行なわれるかまでは指定していないということです。つまり，そのコマンドに応じた処理内容を，しかるべきオブジェクトに指示しなければならないということです。

コマンドを有効/無効にする

あるコマンドを，ある一定の期間，ユーザに使わせたくないことがあります。たとえば，ウィンドウを何もオープンしていない場合，ウィンドウをクローズするための標準コマンドである ***cmClose*** を発行しても何の意味もありません。そこで Turbo Vision では，コマンドを無効にしたり有効にしたりする手段を用意しています。

コマンドのグループを特定して有効にしたり無効にしたりするためには，***TCommandSet*** クラスを使用します。これは，0～255 の数の**セット**をシミュレートするものです (0～255 の範囲にあるコマンドだけが無効にできるので，こうなっています)。 以下は，ウィンドウ関係のコマンドを 2 個無効にする例です。

自然な方法でオーバーロード演算子+=を使用している点に注意してください。**TCommandSet** については第 13 章を参照してください。

```
TCommandSet windowCommands;
  ...
  windowCommands += cmNext;
  windowCommands += cmPrev;
  disableCommands(windowCommands);
```

イベント処理

コマンドを定義し，そのコマンドを発行する何らかのコントロール――たとえばメニュー項目やダイアログボックスのボタンなど――も作成したわけですから，次は，そのコマンドが発生したときにビューに何をさせるかを決める必要があります。

すべてのビューが仮想関数 ***handleEvent*** を継承します。たいていのユーザ入力なら，***handleEvent*** がそれだけで処理を行なうことができます。ビューに，何かアプリケーション固有のことをさせたいのであれば，***handleEvent*** をオーバーライドして，新しい ***handleEvent*** で，次の 2 つのことを定義しておく必要があります。自分で定義した新しいコマンドをどう処理するかということと，マウスイベントおよびキーボードイベントを得たときに，どういう処理を行なわせたいかということです。

ビューの ***handleEvent*** は，ビューがどのような振舞いをするかを決めます。まったく同じ ***handleEvent*** メンバ関数を備えたビューは，まったく同じやり方でイベントを処理します。ビューのクラスを新たに導出する場合，普通は，多かれ少なかれその基本クラスのビューと似た振舞いをさせ，いくらか違ったことをやらせたいわけです。このような場合，ずばぬけて簡単な方法は，新たに導出したオブジェクトの ***handleEvent*** メンバ関数の中で，基本クラスの ***handleEvent*** を呼び出すことです。

導出クラスの ***handleEvent*** は，概略，以下のようなものとなるでしょう。

```
void TNewView::handleEvent( TEvent event )
{
  // 親のビューの処理を変更または排除するためのコード
  TBaseView::handleEvent(event);
  // 付加的な機能を実行するためのコード
}
```

換言すればこうなります。新しいクラスにおいて，何らかのイベントに対する処理を基本クラスとは別の方法で行なわせたい場合には（あるいはまったく処理を行なわせない場合も），基本クラスの ***handleEvent*** メンバ関数に渡す**前に**，そのイベントをトラップす

ればよいでしょう。基本クラスと同じような振舞いをさせたいが，付加機能も持たせたいというのであれば，基本クラスの ***handleEvent*** を呼び出した**後で**，コードを追加してその機能を実現すればよいでしょう。

イベントオブジェクト

この章では，これまでイベントについてかなり理論的に論じてきました。イベントの ***what*** データメンバで指定されるイベントの種類（マウス，キーボード，メッセージ，“なし”）について説明し，またコマンドイベントの ***command*** データメンバの使い方についても簡単に紹介しました。

今度は，イベントオブジェクトの実体について説明することにします。***TEvent*** 構造体は，SYSTEM.H ヘッダファイルで以下のように定義されています。

```
struct TEvent
{
   ushort what;
   union
   {
      MouseEventType mouse;
      KeyDownEvent keyDown;
      MessageEvent message;
   };
   void getMouseEvent();
   void getKeyEvent();
};
```

C++の場合，struct は，単にデフォルトで public アクセスのできるクラスです。

getMouseEvent と ***getKeyEvent*** の 2 個のメンバ関数は，***TProgram::getEvent*** から呼び出されます。プログラムでこれらの関数を直接呼び出す必要はありません。その詳細は第 13 章に示してありますが，簡単に説明しておきましょう。***getMouseEvent*** は，Turbo Vision のイベントハンドラが管理しているマウスイベントキューをポーリングします。マウスイベントが発生していたら，***TEvent::what*** に適当な ***evMouseXXX*** 値を設定します（***XXX*** は ***Down***，***Up***，***Move***，***Auto*** のいずれか)。ボタンと位置の情報が，どのように ***mouse*** データメンバに設定されるかについてはすぐ後で説明します。マウスイベントが何も発生していなければ，***TEvent::what*** に ***evNothing*** を設定します。

getKeyEvent は，DOS や BIOS を呼び出し，キーボードイベントが発生していないか調べます。発生していたら ***TEvent::what*** に ***evKeyDown*** を設定し，***KeyDown*** データメンバに入力された

キーのスキャンコードを記録します。キーボードイベントがない場合には，***TEvent::what*** に ***evNothing*** を設定します。

TEvent::what の値にしたがって，***TEvent*** 内の共用体は，***mouse*** か ***KeyDown*** か ***message*** のいずれかのイベントとして解釈されます。それぞれの種類について，細かく見てみることにします。まずは ***MouseEventType*** ですが，これもまた以下のような構造体として定義されています。

```
struct MouseEventType
{
   uchar buttons;
   Boolean doubleClick;
   TPoint where;
};
```

つまり，マウスカーソルがどこにあり，どのボタンが使用され，ダブルクリックだったかどうか，マウスイベントオブジェクトからわかるということです。

次に，***TEvent::what*** が ***evkeyDown*** だった場合には，***KeyDownEvent*** 構造体からイベントデータを得ることになります。***KeyDownEvent*** 構造体は，以下のような共用体になっています。

```
struct KeyDownEvent
{
    union
        {
           ushort keyCode;
           CharScanType charScan;
        };
};
```

CharScanType も，以下のような構造体になっています。

```
struct CharScanType
{
    uchar charCode;
    uchar scanCode;
};
```

したがって，キーダウンイベントは，***ushort*** 型の ***keyCode*** か ***charCode*** と ***scanCode*** の2個のバイトの組としてとらえることができます。

さて ***TEvent*** の中の共用体の3番目のメンバは ***MessageEvent*** ですが，これは以下のように定義されています。

```
struct MessageEvent
{
    ushort command;
    union
        {
           void *infoPtr;
           long infoLong;
           ushort infoWord;
           short infoInt;
           uchar infoByte;
           char infoChar;
        };
};
```

上のリストにあるように ***MessageEvent*** の共用体には，汎用ポインタあるいは各種の整数値を入れることができます。Turbo Vision では，これらのメッセージの値がいろいろな用途で使用されます。ビューは，自分でイベントを生成して，それを別のビューに送ることができますが，その際にはよく ***infoPtr*** データメンバを利用します。ビュー間の通信および ***infoPtr*** データメンバについては196 ページで説明します。

イベントのクリア

ビューの ***handleEvent*** がイベントの処理を行なう場合，***clearEvent*** メンバ関数を呼び出して処理をしめくくります。***clearEvent*** は，*event.what* を ***evNothing*** にし，*event－＞infoPtr* を this に設定します。これは，イベントが処理済みであることを示す共通の信号です。“なし”イベントが別のビューに渡った場合には，そのビューは，その“なし”イベントを無視しなければなりません。

廃棄イベント

通常は，すべてのイベントがアプリケーション内のどれかのビューによって処理されるはずですが，イベントがどのビューによっても処理されなかった場合には，モードつきビューが ***eventError*** を呼び出します。***eventError*** は，オーナーの ***eventError*** を呼び出

し，同様にして，ビューツリーをさかのぼっていき，最後には，***TApplication::eventError*** が呼び出されます。

TApplication::eventError は，デフォルトでは何も行ないません。プログラムの開発中は，***eventError*** をオーバーライドして，エラーダイアログボックスを表示するなり，ビープを鳴らすなどして役立てることができるでしょう。ただし，エンドユーザからすれば，ソフトウェアがイベントを処理しそこなおうが何ら責任はないわけですから，出荷バージョンにそのようなエラーダイアログボックスを表示させるというのは，おそらく適当な方法ではないでしょう。

clearEvent は，その他，ビュー同士で通信を行なわせる際にも利用することができますが，ここでは，とりあえず，***clearEvent*** を呼び出すまではイベント処理が終わったことにならないということを銘記しておいてください。

イベントメカニズムの変更

モードつきビューが選択されている場合，その中心部では，以下のようなループを行なっています。

```
TEvent event;
   event.what = evNothing;
   do {
      if (event.what != evNothing) eventError(event);
      getEvent(event);
      handleEvent(event);
   } (while endState != continue);
```

イベントの一括取り扱い

イベント駆動型プログラミングがもたらす最大の利益として，イベントがどこから来たのかについては，アプリケーション側で関知する必要がないということがあります。たとえば，ウィンドウオブジェクトの場合，イベントとして ***cmClose*** コマンドを受け取ったときにはクローズしなければならない，ということだけを知っていさえすればよいわけです。そのコマンドが，クローズアイコンのクリックによるものであろうが，メニュー選択によるものであろうが，ホットキーによるものであろうが，あるいはプログラム内のどこか他のオブジェクトからのメッセージによるものであろうが，それは関係のないことなのです。さらには，そのコマンドがそのオブジェクト向けのものなのかどうかについてすら関知する必要はありません。ただ，処理すべきイベントが送られてきたということだけがわかればよいのです。そのイベントの処理方法はわかっているので，あとはそれを実行するだけです。

このような“ブラックボックス”としてのイベントに関して鍵を握っているのが ***getEvent*** メンバ関数です。プログラム内で，イベントの出所に関知する必要があるのは，唯一 ***getEvent*** だけです。アプリケーション内のオブジェクトは，マウスやキーボードのイベントとか，他のオブジェクトが発行したペンディングイベントの読み出

しについては，***getEvent*** を呼び出すだけでよく，あとは ***getEvent*** に任せます。

新種のイベント（たとえば，シリアルポートからの文字の読み出し）を作り出したい場合には，アプリケーションの導出クラスで，単に ***TApplication::getEvent*** をオーバーライドするだけで実現することができます。APP.PAS にある ***TProgram::getEvent*** を見ればわかりますが，***getEvent*** は，マウスイベントとキーボードイベントがないか巡回した後，***idle*** を呼び出すというループになっています。新たにイベントのソースを加えるには，***idle*** をオーバーライドして，シリアルポートから文字を読み出し，それを基にイベントを生成するか，あるいは ***getEvent*** 自体をオーバーライドして，ループに ***getComEvent*** *(event)* 呼び出しを加え，***getComEvent*** で，指定のシリアルポートから 1 文字読み出すことができたら，イベントオブジェクトを返すようにすることで実現できます。

getEvent のオーバーライド

選択されているモードつきビューの ***getEvent*** は，そのオーナーの ***getEvent*** を呼び出します。同様にして ***TApplication::getEvent*** にたどりつくまで，***getEvent*** 呼出しがビューツリーをさかのぼっていきます。実際に次のイベントを取り出すのは，常に ***TApplication::getEvent*** です。

Turbo Vision においてイベントを取り出すのは，いつでも ***TApplication::getEvent*** ですから，イベントに変更を加え，それをアプリケーション全体に有効なものとするには，この ***TApplication::getEvent*** をオーバーライドするだけで済みます。たとえば，キー入力マクロの機能を組み込みたいのであれば，***getEvent*** が返してくるイベントを監視し，あるキー入力が得られたら，それをマクロに展開すればよいわけです。アプリケーションのそれ以下の部分からは，その一連のイベントがユーザからの入力そのものとして送られてきたものとしか見えないわけです。

```
void TMyApp::getEvent(TEvent& event)
{
  TApplication::getEvent(event);
  // { ここで特別な処理を行なう }
```

アイドルタイムの利用

TApplication::getEvent が役割を一手に引き受けていることによる恩恵がもう１つあります。***TApplication::getEvent*** は，イベントが何も発生していない場合には，***TApplication::idle*** を呼び出すということです。***TApplication::idle*** は，ダミーの（空の）メンバ関数であり，それをオーバーライドすることにより，選択されているビューでの処理と並行して，何らかの処理を行なうことができるわけです。

マスターディスクには，サンプルプログラムとして，ヒープビュアーの例が収められています。

たとえば，***THeapView*** と呼ばれるビューを定義し，有効なヒープメモリを表示する ***update*** というメンバ関数をそのビューが使用しているものとします。そのとき，以下のコードで ***TApplication::idle*** をオーバーライドすれば，プログラムがどんな場面を実行していたとしても，ユーザは，有効なヒープメモリを示す表示を見ることができるようになります。

```
void TMyApp::idle()
{
  THeapView::update();
}
```

ビュー間の通信

プログラム内のあるオブジェクトが別のオブジェクトと情報のやりとりを行ないたいとします。従来のプログラミング方法なら，おそらく，単にあるデータ構造から別のデータ構造に情報をコピーすることになるでしょう。しかし，オブジェクト指向プログラムにおいては，オブジェクト同士はお互いがどこに存在するのかを知らないので，そう簡単にはいきません。

ビュー間で通信を行なう必要がでてきた場合，まず最初に考えなければならないことは，2つのビュー間で仕事を適正に配分しているかどうかということです。クラス設計がよくないために，通信を行なう必要がでてきているのかもしれません。たとえば，2つに分けてあるビューは，実は1個のビューにまとめておくべきものだったとか，あるいは一方のビューの一部は，もう一方のビューに入れておくべきものだったというようなことです。

仲介ビュー

プログラム設計は適切であるけれども，それでもビュー同士で通信を行なう必要があるというような場合には，仲介ビューを立てるのが正解かもしれません。

たとえば，スプレッドシートからワードプロセッサに何かペーストしたい，あるいはそれとは反対方向のペーストを行ないたいとします。Turbo Vision のアプリケーションでは，これはビュー間で通信を行なわせることで実現することができます。しかし後になって，たとえばデータベースもこのアプリケーションに加え，データベースとの間でもペーストを行ないたくなったとします。今度は，最初の2つのオブジェクト間で構築した通信機能を拡張して，3つのオブジェクト間で使えるようにする必要がでてきます。

このような場合，仲介ビューを立てるのが旨い解決法となります。上の例の場合，たとえばクリップボードがこれに当たります。そうすれば，オブジェクトは，クリップボードとの間でのコピーやペー

ストの方法を知っておくだけでよいことになります。新たにその通信に加えたいオブジェクトが何個あったとしても，それ以上作業が複雑になるということはありません。

ビュー同士のメッセージ

状況をよく分析してみた結果，確かにプログラム設計は適正なものであり，かつ仲介ビューを立てる必要もないというのであれば，2つのオブジェクトだけの間で簡単な通信を行なわせることもできます。

あるビューが別のビューと通信できるためには，まず相手のビューがどこにあるのかがわからなければなりません。さらに，相手のビューがその時点で実際に存在することまで確認する必要があるかもしれません。

まずは簡単な例から。STDDLG ライブラリには，***TFileDialog*** というダイアログボックスが入っています。***TFileDialog*** では，ユーザにディスクのディレクトリを表示するための ***TFileList*** オブジェクトが使われており，さらに ***TFileList*** の上には，ロードファイルの現在の選択を示す ***TFileInputLine*** オブジェクトが置かれます。この場合，ユーザが ***TFileList*** から別のファイルを選択するたびに，***TFileList*** オブジェクトは，***TFileInputLine*** オブジェクトに対して，新しいファイル名を表示するよう指示する必要があります。

この例では，***TFileList*** と ***TFileInputLine*** は，ともに同じ ***TFileDialog*** オブジェクト内で初期化されるわけですから，***TFileList*** オブジェクトは，***TFileInputLine*** オブジェクトが存在することは確認できます。それでは，***TFileList*** は，ユーザが新しいファイル名を選択したことを，どのようにして ***TFileInputLine*** に伝えているのでしょうか。***TFileList*** はメッセージを作成し，それを送っているのです。その際，イベントの送信には，***TFileList::focusItem*** が使用され，***TFileInputLine*** の ***handleEvent*** がそのイベントを受け取ります。

```
void TFileList::focusItem( short item )
{
  TSortedListBox::focusItem( item );
  // call base member function first
  message(owner, evBroadcast, cmFileFocused, list()->at(item));
}

void TFileInputLine::handleEvent( TEvent& event )
{
    TInputLine::handleEvent(event);
    if( event.what == evBroadcast &&
       event.message.command == cmFileFocused &&
       !(state & sfSelected))
    {
       if( (((TSearchRec *)event.message.infoPtr)->attr & FA_DIREC) !=
           0 )
       {
          strcpy( data, ((TSearchRec *)event.message.infoPtr)->name );
          strcat( data, "¥¥" );
          strcat( data, ((TFileDialog *)owner)->wildCard );
       }
        else
          strcpy( data, ((TSearchRec *)event.message.infoPtr)->name );
          drawView();
    }
}
```

message は，メッセージイベントを生成し，そのイベントを処理するオブジェクトへのポインタを（そういうオブジェクトがあった場合に）返す関数です。

ブロードキャストはどこで処理される？

何か行なう前に，デスクトップ上にオープンしているファイルがあるかどうかを知りたいとしたら，どうすればよいのでしょうか。この場合，ブロードキャストイベントを発行し，そのイベントに対してウィンドウに応答してもらうというのが答となります。そのイベントを処理するオブジェクトは“サイン”を書き残してくれますから，もしそういうウィンドウがあった場合には，どのウィンドウがそのイベントを処理したのかがわかります。この過程は，***TView::clearEvent*** の定義の中に見ることができます。

```
void TView::clearEvent( TEvent& event )
{
    event.what = evNothing;
    event.message.infoPtr = this;
}
```

たとえばビューオブジェクト *v* が ***clearEvent*** を呼び出すことによって，ブロードキャストイベントを処理したことを知らせるとすると，***infoPtr*** には &*v* がセットされるので，それをたどっていくことにより *v* を特定することができます。

誰か外に出てる？

次に，具体的な例を示します。ボーランドの統合開発環境 (IDE) からの例です。ユーザが watch ウィンドウをオープンしようとした場合，プログラムは，すでに watch ウィンドウがオープンしているかどうかを調べる必要があります。なければオープンし，あればそれを最前面にもってくるわけです。

そこで ***message*** 関数を使用してメッセージをブロードキャストします。

```
areYouThere = message(deskTop, evBroadcast, cmFindWindow, 0);
```

watch ウィンドウの ***handleEvent*** は，*cmFindWindow* を受けとると，そのイベントをクリアします。

```
switch (event.message.command)
{
  ...
  case cmFindWindow:
     clearEvent(event);
     break;
  ...
}
```

clearEvent は，前述のように，イベントレコードの ***what*** データメンバに ***evNothing*** を設定するとともに，*infoPtr* に this を設定します。***message*** は，これらのデータメンバを読み出し，イベントが処理済みであれば，そのメッセージイベントを処理したオブジェクトへのポインタを返します。上の例では，メッセージイベントを処理するオブジェクトは，watch ウィンドウとなります。そこで，

ブロードキャストを送出する行に続けて，以下のコードを加えます。

```
if (areYouThere == 0)
  createWatchWindow();  // watch ウィンドウがない場合には、新たに1個作成する
else areYouThere->select;  // すでにあれば、それを前面にもってくる
```

cmFindWindow ブロードキャストに応答できるオブジェクトがwatch ウィンドウのみであるかぎり，ウィンドウをオープンさせるコードは，デスクトップの最前面に watch ウィンドウを確実に1個だけ表示させるようにして，処理を終了することができます。

最前面にいるのは誰？

さらに上述の技法を使用して，たとえば，ウィンドウ型の中でどのウィンドウがデスクトップ上で最前面のビューとなっているかを判断することができます。ブロードキャストイベントは，モーダルビューの各サブビューに，Zオーダー(挿入とは逆の順番)にしたがって送出されるので，一番最後に挿入されたビューが，デスクトップの“一番手前”にあるビューということになります。

今度は，ボーランドのIDEでユーザがデスクトップの最前面にwatch ウィンドウをオープンし，エディタウィンドウ内ではコードをステップ動作させているという状況を考えてみましょう。watch ウィンドウは，アクティブなウィンドウ（太線のフレームをもち，スタックのトップにあるウィンドウ）となることもできるのですが，コードウィンドウの実行バーも，実行中のコードを追跡する必要があります。皆さんがデスクトップに複数のエディタウィンドウをオープンする場合には，それらのウィンドウがまったくオーバーラップしないようにすることもできるでしょうが，統合環境の場合は，どのエディタウィンドウが追跡を行なっているのかを把握しておく必要があります。

もちろん答は，手前にある，すなわち最前面になるエディタウィンドウであり，そのウィンドウが一番最後に挿入されたわけです。どのウィンドウが“一番手前”にあるかを知るために，統合環境は，エディタウィンドウのみが応答することのできるメッセージをブロードキャストします。最初にそのブロードキャストを受け取るのは，一番最後に挿入されたエディタウィンドウのはずであり，そのウィンドウが，イベント処理として，イベントのクリアを行ないます。

したがって，統合環境は，***message*** から戻された結果を読み出すことで，どのウィンドウがコード追跡に使われるのかを知ることができることになります。

handleEvent の呼び出し

以上の方法の他，ビュー間の通信を行なうには，イベントを作成あるいは修正してから，***handleEvent*** を呼び出す方法を用いることもできます。どのビューの ***handleEvent*** であっても呼び出すことができます。呼び出されたビューは，イベントをそれ以下のビューに渡します。したがってイベントはツリーを下っていきます。

ヘルプコンテキスト

Turbo Vision には，アプリケーション内のそれぞれの文脈に合ったヘルプ表示を行なえるようにするツールが組み込まれています。ビューには，ヘルプコンテキスト番号というものを割り当てることができ，Turbo Vision は，そのビューにフォーカスがきたときには，常にそれに割り当てられたヘルプコンテキスト番号が，アプリケーションの現在のヘルプコンテキスト番号となるようにします。

文脈にあったヘルプを一まとめにしたい場合には，皆さんが定義したヘルプコンテキスト番号を管理する ***helpView*** を作成することで実現できます。***helpView*** は，呼び起こされると（通常は，ユーザが ***F1*** か何かのホットキーを押すことで呼び起こされる），***getHelpCtx*** メンバ関数を呼び出すことによって，現在のヘルプコンテキストをそのオーナーに問い合わせることになります。***helpView*** は，それを基に，情況に応じたヘルプテキストを読み出し，表示します。***helpView*** の例が，いくつかデモプログラムの中にあります。

おそらく，文脈に合ったヘルプというのは，アプリケーション開発においては一番最後に作成されるものでしょう。そこで，***TView*** オブジェクトを初期化する際には，デフォルトのコンテキスト ***hcNoContext*** が割り当てられます。これは，定義済みのコンテキストであり，現在のコンテキストを変更しません。やがてヘルプ番号

の割り振りを決めることができるようになったら，ビューをコンストラクトした直後にそのビューの ***helpCtx*** データメンバに適当な番号を設定することによって，番号をビューに割り当てるというわけです。

またヘルプコンテキストは，ステータスラインがどのビューを表示すべきかを決める際にも利用されます。ステータスラインを作成した際に，ある範囲のヘルプコンテキスト値に対応するステータス項目群が ***TStatusDef*** コンストラクタによって作成されたことを思い出してください。新しいビューがフォーカスを得た場合，その項目のヘルプコンテキストが，どのステータスラインを表示すればよいかを決定するわけです。

第6章
信頼性の高いプログラムを開発するには

イベント駆動型の対話型ユーザインターフェースの場合，コマンドラインユーティリティの場合よりもエラー処理がずっと複雑になります。対話型でないアプリケーションの場合，エラーが発生したときにプログラムがエラーメッセージを表示して終了しても，それはそれで済むのですが（また実際，そうなるものだと思われている），対話型の環境の場合，プログラムはエラーからの回復を行ない，ユーザとの対話が続けられるような状態にする必要があります。エラーが発生した場合，それがどういう種類のものであっても，ユーザが扱っていた情報を乱したり，プログラムを終了させたりしないようにする必要があるわけです。こうしたプログラミング規準に合致するものを，“信頼性の高い”プログラムだということができます。

Turbo Vision は，信頼性の高いプログラムを開発するための仕掛を備えています。Turbo Vision のプログラミングスタイルを踏襲することによって，エラー，とくに神出鬼没の“Out of memory”エラーを検出し，エラーから回復するのが容易に行なえるようになります。これは，「単位的」動作という考え方に従って進められるものです。

全か無かのプログラミング

単位的動作というのは，エラー処理に関して，それ以上小さな動作に分解することのできない動作のことです。もっと具体的に言うならば，エラーとなるものであれば絶対にエラーとなり，ちゃんと動作するものであるなら完璧に動作するという，そういう動作のことです。動作を単位的にすることは，とくにメモリ割り当てを行なう際に役に立ちます。

一般に，プログラムでは，メモリを小さなブロック単位で多数割り当てます。たとえば，ダイアログボックスを構築する場合，まずダイアログボックスに対してメモリを割り当て，さらに各コントロールにメモリを割り当てます。これらのいずれの割り当ても失敗に終わる可能性があり，失敗に終わったときには，その次の割り当てに進んでよいのか，そこで割り当てを終わりにするのか判断する必要があります。メモリ割り当てが失敗した場合には，割り当て済みのメモリをどれか解放する必要があります。そこで，すべてのものを割り当ててから，どこかが割り当てに失敗しかたどうかをチェックするというのが望ましい方法です。というわけで，**セーフティプール**の登場となります。

セーフティプール

Turbo Vision では，ヒープの最後の部分に，セーフティプールと呼ばれるある一定量のメモリ（デフォルトでは 8 K バイト）が確保されます。メモリ割り当てがヒープのセーフティプールにまで及んだ場合には，***lowMemory*** 関数が *True* を返してきます。これにより，それ以上メモリ割り当てを行なっても，おそらくちゃんとメモリ割り当てができないであろうということがわかります。

セーフティプールを正しく機能させるためには，プールは単位的動作のうち最大のものに割り当てたメモリ量と同じ大きさにする必要があります。つまりプールは，すべてのメモリ割り当てを ***low-Memory*** でチェックできるような大きさとする必要があります。た

いていのアプリケーションでは，8Kバイトで充分でしょう。

TVMemMgr と ***TBufListEntry*** の2つのクラスが，Turbo Vision の低位のメモリ管理を行ないます。Turbo　Vision では，***new*** 演算子をオーバーライドしているので，メモリ割り当てを行なおうとして際にヒープ上に充分なメモリがなかった場合，***new*** がアボートする前に何らかの手だてを講じることができます。まず，グループの表示処理にキャッシングバッファが使用されている場合には，1番目のキャッシングを解放して再度メモリ割り当てを試みます。それでも失敗した場合には，***new*** は次のキャッシングバッファを解放します。キャッシングバッファを全部解放してもまだメモリ割り当てに失敗した場合には，セーフティプールを調べます。セーフティプールを使いきっている場合には，***new*** は aborts を呼び出します。使いきっていない場合にはセーフティプールを解散し，最後の試みとしてメモリ割り当てを行ないます。それで失敗した場合に，***new*** はアボートします。

従来の難しい方法

メモリ割り当てを従来のやり方でやるとすれば，ダイアログボックスの構築は，以下のようなものとなるでしょう。

```
Boolean OK = True;

TDialog *pd = new TDialog( TRect(20,3,60,10), "My dialog");
if pd != 0
{
    TStaticText *control = new TStaticText(TRect(2,2,32,3),
                          'Do you really wish to do this?');
    if (control !=0) insert(control)
    else OK = False;

    TButton *control = new TButton(TRect(5,5,14,7), '~Y~es', cmYes);
    if (control !=0) insert(control)
    else OK = False;

    TButton *control = new TButton(TRect(16,6,25,7), '~N~o', cmNo);
    if (control !=0) insert(control)
    else OK = False; ·

    TButton *control = new TButton(TRect(27,5,36,7), '~C~ancel',
                                  cmCancel);
    if control !=0 insert(control)
    else OK = False;
}
  if (!OK) destroy(pd);
```

変数 *OK* が，メモリ割り当てで失敗があったかどうかを示します。もし失敗があった場合には，ダイアログボックス全体を廃棄する必要があります。ダイアログボックスを廃棄するということは，そのサブビューも廃棄することになるということに注意してください。

新しい簡単な方法

一方，このコードブロック全体を，セーフティプールに関して単位的な動作とみなせるようにするためには，以下のように変更します。

```
TDialog *pd = new TDialog( TRect(20,3,60,10), "My dialog");
TStaticText *control = new TStaticText(TRect(2,2,32,3),
                        'Do you really wish to do this?');
TButton *control = new TButton(TRect(5,5,14,7), '~Y~es', cmYes);
TButton *control = new TButton(TRect(16,6,25,7), '~N~o', cmNo);
TButton *control = new TButton(TRect(27,5,36,7), '~C~ancel',
                               cmCancel);

if (lowMemory())    // セーフティプールに入り込んだかどうかのチェック
{
  destroy(pd);
  outOfMemory();    // out-of-memory エラーの報告
  doIt = False;
}
else
  doIt = (desktop->execView(pd) == cmYes);
```

```
doIt = (validView(pd) != 0) && (desktop->execView(pd) == cmYes);
```

も必要ありません。したがってセーフティプールは充分な大きさですから，このコードですべてのメモリ割り当てをちゃんと行なうことができることになります。ダイアログボックスのメモリ割り当てを全部行なったら，***lowMemory*** をチェックします。*True* になっていれば，ダイアログボックス全体を廃棄します。*True* でなければ，このダイアログボックスを使用できることになります。

validView メンバ関数

lowMemory のチェックは，たびたび行なう必要があるので，***TApplication*** は，必要なチェックを行なうための ***validView*** というメソッドを備えています。***validView*** を利用すれば，上のコードの最後の 8 行にある ***lowMemory*** の検査は，以下の 2 つにまとめることができます。

```
doIt = (validView(pd) != 0) && (desktop->execView(pd) == cmYes);
```

validView は，引数として渡されたビューへのポインタを返します。ただし，そのビューが無効であった場合には，0 が返されます。***validView*** は，***lowMemory*** が ***True*** を返してきたら，そのときのビューを廃棄し，***outOfMemory*** を呼び出します。

delete と destroy

上のコードでダイアログオブジェクトを消去するのに，従来の C++ の delete 演算子ではなく，***destroy*** (***pd***) を呼び出しているのに気付いた方もいらっしゃるでしょう。Turbo Vision では，多くのクラスやオブジェクトが相互に依存しているので，***TObject*** から導出されたオブジェクトは，Turbo Vision に固有な方法で消去する必要があります。このために ***TObject*** には，引数として ***TObject*** へのポインタだけをとる ***destroy*** というメンバ関数が用意されています。***TObject*** から直接または間接的に導出されたオブジェクトを消去する場合には，通常の delete 演算子ではなく，この ***destroy*** を呼び出す必要があります。

destroy 自身もそうでしたが，舞台裏で ***destroy*** は ***shutDown*** という仮想関数を呼び出します。***TObject*** のメンバ関数 ***shutDown*** は，***TObject*** の導出クラスの多くでオーバーライドされ，それぞれのクラスに応じたシーケンスでオブジェクトの消去を行なうようにされています。たとえば ***TGroup::shutDown*** は，そのサブビューごとに消去とバッファの解放を繰り返し，最後に

TView::shutDown を呼び出してそのビュー自身を消去します。ただし Turbo Vision のメモリ管理システムをあえて改造して高度なものにするつもりでなければ，この機構に関与する必要はありません。通常のアプリケーションなら，いつもどおり new を使用し，***TObject*** から導出したオブジェクトを直接廃棄する場合には，delete ではなく ***destroy*** を使う必要があるということだけを理解しておいてください。

メモリエラー以外のエラー

当然ながら，メモリエラーだけがエラーだというわけではありません。たとえば，ディスクファイルから何らかの情報を読み出すために，あるビューを呼び出した場合に，そのファイルが存在しなかったとか，無効なファイルであったということが起こりえます。この種のエラーも，やはりユーザに知らせる必要があります。そのような状況を踏まえて，***validView*** には，メモリエラー以外のエラーを処理するための“フック”も組み込まれています。つまり，***validView*** はそのビューの ***valid*** メンバ関数を呼び出すようになっています。

TView.valid は，デフォルトで *True* を返してくるようになっています。***TGroup.valid*** は，そのグループの管理下にある**すべての**サブビューの ***valid*** 関数が *True* を返してきたときのみ，*True* を返してきます。つまりグループは，そのサブビューがすべて有効である場合に有効だということです。メモリエラー以外のエラーが起こる可能性のあるビューを作成する場合には，そのビューの ***valid*** をオーバーライドして，そのビューがちゃんとインスタンス化できたときのみ *True* を返すようにする必要があります。

valid は，あるビューが何らかの理由で使用できないことを示すために使うことができます。たとえば，ビューが自分自身で管理しているファイルを探し出すことができなかったというような場合です。ただし，***valid*** が何をどのようにチェックするかは，プログラマのみなさんが決めることです。一般的な ***valid*** 関数は，だいたい以下のようなものとなります。

```
Boolean TMyView::valid(ushort command)
{
   if ( command == cmValid && errorEncountered )
   {
      reportError();
      return False;
   }
   else
      return True;
}
```

ビューを最初に実体化する際には，***command*** 引数に ***cmValid*** を指定して ***valid*** メンバ関数を呼び出し，そのビューの作成に当たってメモリエラー以外のエラーがなかったかどうかをチェックする必要があります。***validView(X)*** では，セーフティプールのチェックを行なうとともに，***X::valid(cmValid)*** も自動的に呼び出します。そこで，ビューを新たに使用する際には，その前に ***validView*** を呼び出せばよいということになります。

valid は，また，モードつき状態が終了するときにも必ず呼び出されます。その際，***command*** 引数には，そのモードつき状態を終了させたコマンドが指定されます（第4章『**ビュー**』を参照）。これによって，アプリケーションを終了する際にエディタウィンドウのテキストがセーブされていないなどの条件でトラップをかけることができるようになります。

errorEncountered は通常 ***Boolean*** 型のデータメンバとし，プログラムの必要に応じて指定や設定を行なうことになります。一般に *TMyView* コンストラクタは，***errorEncountered*** を *False* に初期化し，アプリケーション独自のエラーチェックで *True* にします。

エラーの報告

エラーが発生した場合には，ビューは画面には現われないわけですから，***valid*** メンバ関数が *False* を返す前に，どのようなエラーであれ，ユーザに知らせる必要があります。上の例でこれを行なっているのが ***reportError*** 呼び出しです。通常は，メッセージダイアログボックスをポップアップさせるなどします。このような場合，

どのようなエラーであれ，エラーを知らせる義務は個々のビューにあります。したがって，プログム自体は個々のビューで起こり得る状態に関知する必要はないわけです。

これによって，何かうまくいっていないことがないかを常に監視するのではなく，すべてはうまくいっているものとしてプログラムを組むことができるわけですから，これはプログラミグ技法上の大きな進歩だということができます。アプリケーションなどのグループオブジェクトは，エラーのチェックに手を煩わす必要がまったくないわけです。ただし，管理下にあるビューが無効でないかどうかはチェックする必要があります。何か無効なサブビューがあった場合でも，グループは，そのグループ自身およびその管理下のサブビューを廃棄し，オーナーにそのグループが無効であることを知らせればよいだけです。**グループは，無効なサブビューがどういうトラブルがあったのかをユーザにすでに知らせたいものとみなすことができるわけです。**

valid を利用することによって，ウィンドウやダイアロクセボックスを単位的動作として構築することができます。ウィンドウを表示する各サブビューを構築する際に，エラーがあったかどうかをチェックする必要はありません。コンストラクタが構築に失敗した場合には，単にその ***valid*** が *False* を返してくるだけです。その場合，ウィンドウは最後まで構築を進め，その時点でウィンドウ全体を ***validView*** に渡すことができます。ウィンドウ内に無効なサブビューがあった場合には，ウィンドウ全体がチェックの結果を *False* として返します。そうすると，***validView*** は，そのウィンドウを廃棄し，0 を返します。やらなければならないことは，***validView*** から返されてきた結果をチェックすることだけです。

大量消費ビュー

valid 関数は，また，メモリの**大量消費**ビューについても処理することができます。大量消費ビューというのは，ファイルをまるごとメモリにロードしてくるような，セーフティプールよりも大きなメモリサイズの割り当てを行なうビューのことです。大量消費ビューの場合は，自分自身で ***lowMemory*** をチェックする必要があり

ます。構築がすべて終わるまで待ち，構築が終わった時点で ***validView*** にチェックしてもらうというわけにはいきません。

大量消費ビューが，自分自身を構築している最中にメモリを使いきってしまった場合には，それ以上メモリを割り当てないようにするためにデータメンバとして設けてあるフラグ（先の例の ***error-Encounterd*** フラグのようなもの）を立ててエラーが発生したことを示す必要があります。フラグは ***valid*** でチェックされ，ビューは，*MyApplicalion－＞outOfMemory* を呼び出し，***valid*** 呼び出しの結果として *False* を返すでしょう。

これは確かに，コンストラクタが正しく働いているとみなせることに比べれば，あまりエレガントだとはいえませんが，セーフティプールのサイズを超えるビューを構築するには，これ以外には手はありません。

マスターディスクに入っている例題プログラム FILEVIEW.CPP では，これらの技法を使って安全なファイルビュアーを作成しています。たとえば，***TFileViewer*** コンストラクタが ***isValid*** を *True* にします。***readFile*** 呼び出しが失敗した場合には，***mes-sageBox*** が "Invalid drive or directory（無効なドライブまたはディレクトリ）"というメッセージを表示して ***isValid*** を *False* にします。

第7章
コレクション

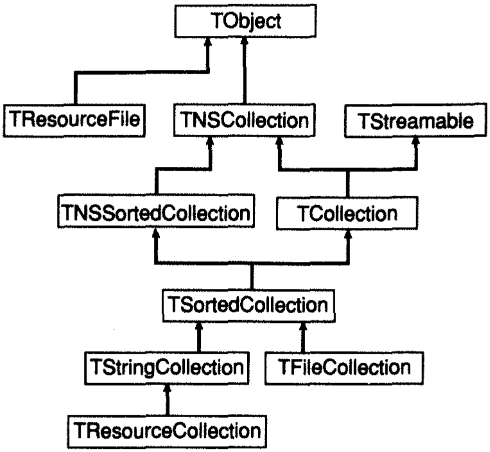
従来のCの配列には，おもに2つの制約がありました。まず，初期化された配列のサイズはコンパイル時に決定されるということ，そして第2に，各要素のデータ型が1つの配列に対して同じでなくてはならないということです。後者の制約は，汎用のポインタを要素とする配列を使うことで回避できますが，型の不一致が起こらないようにするための手だてが必要です。Turbo Vision では，**コレクション**クラスをいくつか用意することにより，この2つの問題を解決しています。コレクションとは，動的にサイズを変更することのできる，任意のオブジェクトからなる汎用的な配列であると考えられます。また，後で説明しますが，コレクションを**ストリーム入出力可能**とすることもできます。つまりコレクションは，型の保障をかなり厳密に行ないながら，出力ストリームに書き込んだり，入力ストリームから読み込んだりすることもできるということです。クラスをストリーム入出力可能なものとすれば，数々の大切なアプリケーションで使用されるオブジェクトを**持続性のある**ものとすることができます。たとえば，どのようなグループ（場合によってはアプリケーション全体）であっても，2，3 行プログラムを追加することによって，そのすべてのサブビューを保存したり復元したりすることができるようになります。

TNSCollection（ストリーム入出力できないコレクション）および ***TCollection***（ストリーム入出力可能なコレクション）を基本クラスとして，特殊用途のクラス（ソート済みのコレクション，ファイルのコレクションなど）をいろいろ導出することができます。ストリーム入出力可能なコレクションをさらに特殊化したのが**リソース**と呼ばれるクラスであり，文字列（名前）をインデックスとして任意のオブジェクトを保存したり復元するために使用されます。下

図は，コレクションとリソースのクラス階層を示したものです。

図 7.1
コレクションとリソース

C++ の慣例に従い，矢印は基本クラスの方向を指しています。



TCollection クラス

TNSCollection クラスは，オブジェクトのコレクションを保存したり処理するための基本的な機構を用意します。***TNSCollection*** をストリーム入出力可能にした導出クラスが ***TCollection*** です。***TCollection*** は，***TNSCollection*** と ***TStreamable*** を多重継承しており，アクセスと保存の機能については ***TNSCollection*** から継承し，一方「ストリーム入出力可能性」については，ある部分は ***TStreamable*** から，別の部分を 2 つの private の純粋仮想メンバ関数 ***TCollection::readItem*** および ***TCollection::writeItem*** から継承しています。***TStreamable*** のほうは，各導出クラスが，それぞれのデータ型に合わせて基本的なストリーム入出力を行なえるようにオーバーライドする必要がありますが，これについては第 8 章で説明します。

以下では，おもにコレクションのクラスおよびオブジェクトについて説明しますが，これは，***TCollection*** か ***TNSCollection*** から導出されたクラスおよびオブジェクトのことを意味します。

TNSCollection, ***TCollection***, および ***TStreamable*** は，基本クラスとして，なんらかの用途を持つ下位クラスを導出するためだけに作られたクラスです。実際，***TCollection*** と ***TStreamable*** はいずれも抽象クラス（それぞれ少なくとも 1 個の純粋仮想関数を持つ）であり，***TCollection*** または ***TStreamable*** のオブジェクトを直接インスタンス化することは**できません**。***TNSCollection*** は，厳密には抽象クラスではないので，TVGUID17.CPP に見られるように，そのインスタンスを作成することもできますが，そのようなオブジェクトを実際のアプリケーションで使用する場合には一定の制約があります。

TNSCollection クラス（*NS* は，non-streamable，ストリーム入出力不可の意）は，***TObject*** の導出クラスであり，コレクションの処理に必要な基本的なメンバをすべて備えています。基本的なメンバとしては，コレクション要素のアクセス，追加，および削除を行なう関数，並びに 3 種類の**繰り返し処理関数**などがあります。繰り返し処理関数とは，コレクション中の全要素または選択した要素

に，プログラマが独自に定義した関数を適用するための関数です。一方，***TStreamable*** は，ストリームに名前を付けたり，ストリームの登録を行なうための抽象クラスです。**opstream** と **ipstream** の2つのクラスは ***TStreamable*** のフレンドなので，おなじみのC++ のストリーム演算子 >> と << を適当にオーバーロードして使用することができます（詳しくは第 8 章を参照してください)。

このように ***TNSCollection*** と ***TStreamable*** を継承している ***TCollection*** は，ストリーム入出力可能なコレクションの機能を用意するわけですが，この章では，***TNSCollection*** から継承しているストリーム入出力可能でない部分について重点的に説明を行ないます。コレクションのストリーム入出力を行なわない場合でも，***TNSCollection*** ではなく ***TCollection*** を基本クラスとする場合には，通常，オーバーヘッドが少し発生します。***TCollection*** などのストリーム入出力可能なクラスからの導出クラスは，ストリーム入出力可能なクラスとしての機能を持たせるために，やはり ***TStreamable*** からの継承も行なっているという点に注意してください。

動的なサイズ決定

コレクションオブジェクトに格納されている要素数は，データメンバ ***count*** で示されますが，プログラムの実行中に動的に変更することができます。サイズの拡張は，次のようにして行なわれます。サイズの初期値 ***limit***, および増分値 ***delta*** を設定します。指定してある限界を越えて，要素の追加が行なわれた場合，コレクションの最大許容サイズは ***delta*** 分だけ増やされます。したがって，***delta*** をゼロに設定すれば，コレクションのサイズを固定にすることになります。

コレクション要素の型の混在

TNSCollection の protected のデータメンバ ***items*** は，void

items として宣言されています。このように汎用のポインタからなる配列を使用することにより，任意のサイズの任意の要素（オブジェクトまたは非オブジェクト）を，任意の個数だけ要素とする効率的なコレクションを作成することができます。コレクションオブジェクトに複数のデータ型を混在させる場合，型の不一致が起こらないようにするのはプログラマの責任です。*TNSCollection*** オブジェクトは，渡された要素の内容については「何も知りません」。***TNSCollection*** オブジェクトは，単に要素を格納し，要求に応じてそれを返すだけの働きしかしません。もちろん導出クラスにおいて特定のオブジェクトを処理するために，要素のポインタをキャストしなおすこともできます。とくに複数の型を混在させているコレクションの場合，導出したコレクションクラスでは，***atPut***（指定したインデックスの位置に要素を追加する）や ***at***（指定したインデックスの位置に要素を返す）などのアクセス関数をオーバーライドして，型チェックを行なうようにする必要があります。

コレクションの作成

コレクションの作成は，登録しようとするデータ型を定義するのと同じぐらい簡単に行なうことができます。たとえば，何かのコンサルタントをやっていたとして，顧客の口座番号，名前，電話番号を登録したり，読み出したりしたいとします。そのような場合，まずコレクションに登録するオブジェクトとして，顧客の構造体クラス（***Tobject*** からの導出クラス *TClient*）を定義します。次に，*TClient* のコンストラクタを作成します。***TObject::~TObject*** が顧客の廃棄を行なってくれるので，デストラクタを明示的に定義する必要はありません。

```
struct TClient : public TObject
{
// デフォルトではすべてのメンバがパブリック

// データメンバ 3 個
    const char *account;
    const char *name;
    const char *phone;

    TClient( char *newAccount, char *newName, char *newPhone);
    ~TClient();
```

```
};

// コンストラクタ
TClient::TClient( char *newAccount, char *newName, char *newPhone) {
    account = newStr( newAccount );
    name = newStr( newName );
    phone = newStr( newPhone );
};
```

後は，コレクションをインスタンス化して，それに顧客レコードを挿入していけばよいわけです。プログラムの本体部分は，次のようなものとなります。

```
TNSCollection clientList( 50, 10 );    // limit = 50, delta = 10

clientList.insert( new TClient("90-167", "Smith, Zelda", "(800)
                              555-1212" ));
clientList.insert( new TClient("90-160", "Johnson, Agatha", "(302)
                              139-8913" ));
clientList.insert( new TClient("90-177", "Smitty, John", "(406)
                              987-4321"));
clientList.insert( new TClient("91-100", "Anders, Smitty", "(406)
                              111-2222"));
```

PrintAll と **SearchArea** については，223 ページで説明しています。

```
printAll( &clientList );
searchArea( &clientList, "(406)" );
return 0;
```

コレクションの作成が簡単だということがおわかりになると思います。最初の文では，*clientList* という新しい ***TNSCollection*** オブジェクトを，顧客 50 人分（データメンバ ***limit***）のサイズを初期値として与えて作成しています。*clientList* に 50 人以上挿入する場合には，必要に応じて，サイズは 10 人分単位（データメンバ ***delta***）で大きくなっていきます。次の文では，新しい顧客オブジェクトを作成して初期化し，それらのオブジェクトへのポインタをコレクションに挿入しています。

insert メンバ関数は仮想関数であり，それぞれの用途に応じて（通常は，型を保障した処理を行なうようにするために）オーバーライドすることができます。***insert*** は，デフォルト処理として要素をコレクションの末尾に追加し，***count*** データメンバをインクリメントします。また ***insert*** は，***limit*** に達した場合に，前述した ***delta*** 個分の拡張の処理も行ないます。そして ***insert*** は，新しい

要素の ***index*** を返します。インデックスの型は ***ccIndex*** です。***TNSCollection*** メンバ関数の多くが，引数にインデックスを使用しています。***ccIndex*** は，現在 int に typedef されていますが，将来の移植性のことを考えて，必ず ***ccIndex*** を使用するようにしてください。たとえば，メンバ関数 ***at*** は，

```
void *at( ccIndex index );
```

と宣言されており，***index*** の位置の要素へのポインタを返します。この逆の操作を行なうのが以下の ***indexOf*** であり，指定された要素のインデックスを返してきます。

```
virtual ccIndex indexOf( void *item );
```

insert と同類のメンバ関数に ***atInsert*** がありますが，こちらは，指定したインデックスの位置に要素を挿入します。***remove*** と ***atRemove*** は，同じ機能を果たすメンバ関数であり，コレクションから要素を削除し，それにしたがってインデックスを調整します。***removeAll*** メンバ関数は，すべての要素を削除し，***count*** を 0 にします。これらの「削除」メンバ関数は，格納されている要素を削除するだけで消去するわけではありません。***free*** および ***freeAll*** が要素を削除し，**かつ**消去を行なうためのメンバ関数です。

プログラムの終了時にコレクションが消去される場合，通常，顧客をはじめとするコレクション全体も消去されます。この処理は，***Boolean*** 型のデータメンバ ***shouldDelete*** で制御されます。***shouldDelete*** が *True*（デフォルト）の場合，***freeAll*** を呼び出してすべての要素の削除と消去が行なわれ，*False* の場合はコレクションの ***limit*** を 0 にしますが，要素自体は消去されません。コレクションはポインタからなる配列なので，要素へのポインタを削除することと要素自体を消去することには大きな違いがあります。

繰り返し処理メンバ関数

要素を挿入したり削除したりするだけがコレクションに共通する処理だというわけではありません。コレクションのオブジェクト**全部**について，データを表示したり，何らかの計算をする場合など，**for** ループ処理をさせたいことがよくあります。また，コレクションの中で，ある検索規準に合致する最初の要素と最後の要素を引き出してきたい場合もあります。このようなことを行なうために，コレクションでは ***forEach***，***firstThat***，および ***lastThat*** の 3 つの**繰り返し処理**メンバ関数を用意しています。いずれも引数として，関数ポインタと汎用ポインタ void*arg の 2 つをとります。汎用ポインタをとることにより，メインとなる繰り返し処理関数以外に任意の引数を渡すことができます。

forEach 繰り返し処理メンバ関数

forEach は，以下のように，***ccAppFunc*** 型の *action* と，汎用ポインタ *arg* の 2 つの引数をとります。

```
void TNSCollection::forEach( ccAppFunc action, void *arg)
{
    for( ccIndex i = 0; i < count; i++ ) // count は,
                                         // コレクション中の
                                         // 要素の数
        action( items[i], arg );
}
```

処理関数の型である ***ccAppFunc*** は, 以下のように定義されています。

```
typedef void (*ccAppFunc)( void *, void *);
```

処理関数の最初の引数は，コレクション中の要素へのポインタであり，2 番目の引数は，***forEach*** に渡される汎用の *arg* ポインタです。繰り返し処理メンバ関数が補助的なデータを必要としない場合には，2 番目の引数は 0 にします。***forEach*** は，その処理関数を

コレクションの各要素ごとに1回ずつ，要素が格納されている順番に従って，呼び出します。TVGUID17の *printAll* 関数では，この ***forEach*** 繰り返し処理関数を使用しています。

```
static void print( void *c, void * )
// 汎用ポインタを使用していることから，安全性を考えて static にしている
// 第2引数は使用しない
{
    TClient *tc = (TClient *)c;
    cout << setiosflags( ios::left )
         << setw(10) << tc->account
         << setw(20) << tc->name
         << setw(16) << tc->phone
         << endl;
}

void printAll( TNSCollection *c )  // すべてのクライアントの情報を出力
{
    cout << endl << "Client List:" << endl;
    c->forEach( &print, 0 ); // クライアントごとに print を呼び出す
}
```

forEach に渡された *&print* は，***ccAppFunc*** データ型となっています。すなわち，引数に(void*, void*)をとり，void を「返す」関数へのポインタとなっています。また ***TNSCollection*** オブジェクト **c* の各要素ごとに *print* 関数を呼び出し，それぞれの顧客情報を表示しています。

繰り返し処理関数 firstThat，lastThat

コレクション内のすべての要素に対してある手続きを実行すること以外にも，コレクション内のある特定の要素を何らかの規準に照らして見つけだすことが必要になることもよくあります。繰り返し処理関数 ***firstThat*** および ***lastThat*** は，このような処理を行なうためのものです。名前からわかるように，引数として渡された ***Boolean*** 型の *test* 関数の条件に合致する要素をコレクションから検索します。検索方向は，***firstThat*** と ***lastThat*** とで反対になっています。

firstThat および ***lastThat*** は，検索条件に合致する最初の(もしくは最後の)要素に対するポインタを返します。上の顧客リストの例で，ある顧客の口座番号が，あるいは正確なラストネームが思い出

せなかったとします。幸い，この顧客がモンタナ州で最初に獲得した顧客であったことを憶えていました。そこで，エリアコード 406 における一番最初の顧客を見つけだせばよいということになります（この場合は，リストが日付順になっていたということで）。以下が，***firstThat*** メンバ関数を利用してこの検索を行なう関数です。

```
Boolean areaMatch( void *c, void *ph )
{
    char *areaToFind = (char *)ph;
    TClient *tc = (TClient *)c;
    // 最初の 5 文字で一致部分を検索 "(xxx)"
    if( strncmp( areaToFind, tc->phone, 5 ) == 0 )
        return True;
    else
        return False;
}

void searchArea( TNSCollection *c, char *areaToFind )
{
    TClient *foundClient =
        (TClient *)(c->firstThat( &areaMatch, areaToFind ));
    if( !foundClient )
        cout << "No client met the search requirement" << endl;
    else
        {
            cout << "Found client:" << endl;
            print( foundClient, 0 );
        }
}
```

firstThat に渡すテスト関数へのポインタの型は以下のとおりです。

```
typedef Boolean (*ccTestFunc)( void *, void *);
```

返される型を除き，テスト関数は ***forEach*** で使われた ***ccAppFunc*** 型と同じパターンです。*areaMatch* は 2 番目の *arg* 引数を使って対象となるエリアコードの文字列を渡し，この文字列が顧客のエリアコードを示す *tc−>phone* と一致した場合のみ *True* を返します。コレクション内のどのオブジェクトも検索条件に合致しない場合には，0 ポインタが返されますので，if (! foundClient) というテストとなります。

forEach は ***ccAppFunc*** 型の関数ポインタをとるのに対し，***firstThat*** と ***lastThat*** は ***ccTestFunc*** 型の関数ポインタをとるという点を憶えておいてください。どちらの場合でも，ユーザ定

義の処理関数，テスト関数は，2個の汎用ポインタをとります。1つはコレクション中の要素へのポインタであり，もう1つはそれとは別に繰り返し処理関数に渡したいデータへのポインタです。みなさんはこれらのポインタをキャストしなおして，コレクションで使用される実際の要素と一致させる必要があります。

TVG17B.CPP では，コレクションの処理をさらに信頼性の高いものとするための方法を示しています。つまり，***TNSCollection*** から *TClientCollection* という顧客コレクション用のクラスを導出しています。

これは TVG17B.CPP です。

```
class TClientCollection : public TNSCollection
{

public:

    TClientCollection (ccIndex aLimit, ccIndex aDelta ) :
          TNSCollection( aLimit, aDelta ) {}

    virtual ccIndex insert( TClient *tc )  { return
           TNSCollection::insert( tc ); }

    void printAll();
    void searchArea( char *areaToFind );

};
```

このようにして，***TNSCollection::insert*** (*void *item*) をオーバーライドすることにより，型保障の信頼性を高めることができます。基本クラスの ***insert*** が，どのような型に対するポインタであっても受け入れるのに対して，新しい ***insert*** は引数に *TClient* へのポインタしか受け付けないことになります。アプリケーションをさらに完全なものとする場合には，***at***，***indexOf***，***remove***，***free*** などもオーバーライドし，引数に *TClient* へのポインタをとるようにします。専用の顧客クラスを作成したので，先ほどのコードのようにメンバ関数として *printAll* と *searchArea* を作成することもできます。TVG17B.CPP は，必要な変更は行なっていますが，基本的には TVGUID17.CPP と同じことを行なっています。

ソート済みコレクション

場合によっては，データを何らかの順番に従って並べておきたい場合があります。Turbo Vision では，任意の方法でデータをソートさせることのできる 2 つの特殊な基本コレクションクラスとして，***TNSSortedCollection*** と ***TSortedCollection*** が用意されています。

TNSSortedCollection は，***TNSCollection*** の導出クラスであり，与えられたオブジェクトを自動的にソートします。また，メンバを新たに追加する場合には，自動的にコレクションをチェックし，メンバが重複しないようにします（**Boolean** 型のメンバ **duplicates** を *True* にしない場合）。

TSortedCollection は，***TNSSortedCollection*** をストリーム入出力可能にしたクラスであり，***TNSSortedCollection*** と ***TStreamable*** を基本クラスとしています。***TNSCollection***（ストリーム入出力不可）と ***TCollection***（ストリーム入出力可）に関して前述したことは，ソートを施したクラスについてもそのまま当てはまります。以下では，おもにソート済みのコレクションについて説明しますが，冗長にならないように，ストリーム入出力可能であるか否かについては断らないことにします。

ソート済みのコレクションが違っている点は，***TNSSortedCollection*** と ***TSortedCollection*** がともに抽象クラスであるということです。これらのクラスを使用する場合には，まずコレクションにしたいデータの型を決め，次にそのコレクションにあったソートの仕様を満足するメンバ関数を 2 個定義する必要があります。前回と同様，ここでもストリーム入出力可能かどうかということは問題とせずに ***TNSSortedCollection*** の例を使用することにします。***TNSSortedCollection*** から，*TSortedClientCollection* という新しいコレクション型を導出してみましょう。

TSortedClientCollection は，コレクションがどのような機能のものであるかについてはすでに知っています。新しい顧客レコードを ***insert*** することも，既存のレコードを ***delete*** することもできます。このような基本的な機能は，すべて ***TNSCollection*** から継承

しているわけです。*TSortedClientCollection* で定義しなければならないことは，ソートキーにどのデータメンバを使うかということと，2人の顧客の比較方法，すなわちどちらをコレクション内で前の方にもってくるかということです。このためには，***keyOf*** と ***compare*** の各メンバ関数をオーバーライドします。ところで，***compare*** メンバは，private の純粋仮想関数なので，必ず再定義しなければなりません。以下は，*TSortedClientCollection* クラスの定義の一部です。

```
class TSortedClientCollection : public TNSSortedCollection
{

public:

   virtual void *keyOf(void *item);

private:

   virtual int compare( void *key1, void *key2);
}

void *TSortedClientCollection::keyOf(void *item)
{
   return ( ( (TClient*)item )->name );
}

int TSortedClientCollection::compare(void *key1, void *key2)
{
   return (strcmp ( (char *)(key1), (char *)(key2) ));
```

void* ポインタは，型キャストする必要があります。

KeyOf では，どのデータメンバ(複数指定可)をソートキーとして使用するかを指定します。上の例では，顧客の *name* データメンバにしています。***compare*** では，2つのソートキーへの汎用ポインタをもらい，ソート順でみたとき，どちらを先にもってくるかを決めます。***compare*** は，$key1 < key2$ のとき -1を，$key1 = key2$ のとき 0を，$key1 > key2$ のとき1を返します。上の例では，キー(*name*)の文字列でアルファベット順のソートを行なうようにしています。

keyOf から返されたり，***compare*** に渡されたりするキーは，void＊ポインタですので，ポインタを逆参照する場合には，***char**** キャストする必要があります。

指定しなければならないのはこれだけです。後は，基本クラスを ***TNSSortedCollection*** にして，*TClientCollection* のかわりに *TSortedClientCollection* を指定することで，TVG17B.CPP の改造

は終わりです。これにより，以下のようにして，顧客をアルファベット順にリスト表示させることができるようになります。

これは，TVGUID18.CPP です。

```
int main()
{

   TSortedClientCollection clientList( 50, 10);
   ...
// 以前のものと同じ

}
```

名前のかわりに口座番号で顧客リストをソートしたい場合でも非常に簡単に行なうことができます。***keyOf*** メンバ関数を変更して，*name* のかわりに *account* データメンバを返すようにすればよいだけです。

文字列コレクション

ソート済みの文字列を登録したい場合が多々あります。そのような場合のために，Turbo Vision では特別な用途のコレクション ***TStringCollection*** を用意しています。***TStringCollection*** の各要素は，文字列へのポインタ（char ** 型）です。文字列コレクションは，***TSortedCollection*** の導出クラスなので，データメンバ ***duplicates*** が *True* にされているときのみ，文字列の重複が許されます（デフォルトは *False* で，この場合重複は許されません)。

文字列コレクションは，その他のソート済みコレクションと同じようにして使用することができます。次の例では，***TStringCollection*** から *TWordCollection* を導出し，コンストラクタと *print* メンバ関数を指定しています。*print* メンバ関数は，後述の繰り返し処理関数（これについては後述します）を使って，単語コレクション全体を表示します。

これは，**TVGUID19.CPP** です。

```
class TWordCollection : public TStringCollection
{

public:

   TWordCollection( short aLimit, short aDelta ) : TStringCollection(
        aLimit, aDelta ) {}
   virtual void print();
};
```

ソート済みの単語コレクションは，テキストファイルを 1 行ずつ読み込み，その行から各単語を取り出して，それらの単語を *TWordCollection* オブジェクトにアルファベット（ASCII）順に挿入していくことで構築されます。*insertWord* 関数は，単語が句読点またはホワイトスペースで囲まれた句読点またはホワイトスペース以外の文字の連続であるという，非常に単純化した考え方に基づいて各行を解析していきます。さらに *insertWord* は，空でない単語を検出した場合，その挿入も行ないます。

TWordCollection は，ソート済みのコレクションクラスなので（***TStringCollection*** と ***TSortedCollection*** を介して)，単語文字列が重複してもよいかどうかを決めるための ***duplicates*** デー

タメンバを持っています。上の例では，***duplicates*** は *False* にされているので（これはデフォルトなので，明示的な処理は必要ありません），コレクションは一義的な単語だけを受け入れます。***duplicates*** データメンバは，***TNSSortedCollection*** から継承した ***search***，***indexOf***，および ***insert*** の各メンバ関数の処理に影響を与えます。***insert*** が呼び出されると，コレクション内で対象となる要素が検索されます。その要素が見つからなかった場合には，ソート状態を乱さないように，その要素を必ず正しい位置に挿入します。その要素が見つかった場合に，次に何を行なうかが，***duplicates*** の値によって決まります。***duplicates*** が *True* の場合には，要素をそれと同じ値を持つ要素の前に挿入し，*False* の場合には挿入を行ないません。

単語コレクションは，**main** 内で次のようにして作成されます。

```
TWordCollection *wordCollection = new TWordCollection( 50, 5 );
```

したがって *wordCollection* は，最初に 50 個の文字列ポインタを持ち，以後は 5 個ずつ拡張していくことになります。*fileRead* 関数は TVGUID06.CPP で使われているものに似ていますが，ここでは一度に 1 行だけを読み込み，各行を読み込んだあとで，*insertWord* を呼び出します。***insert*** 呼び出しの中で，以下のように，***newStr*** を使用している点に注意してください。

```
sc->insert( newStr( wstr ));
```

これを，もっと「明確な」（しかも完全に正しい）次の文と比較してみてください。

```
sc->insert( wstr );
```

newStr グローバル関数は，*str*，つまり今読み出してきた単語文字列のコピーを作成しますが，この文字列のコピーのアドレスがコレクションに渡されます。なぜこのようにするかというと，コレクションに対する割り当てメモリが解放されたときに，「元の文字列」ではなく，コピーされた文字列とそれへのポインタが消去されるようにするためです。上の例では，2 つの ***insert*** の違いに大した意味はありませんが，別の状況でとても重要な意味を持つことがあります。もとの文字列が消失する場合には，コピーを保存する必要があ

ります。コレクションは，文字列をコピーすることによってその文字列を管理するのです。

単語コレクションが完成したら，*print* を使用して，コレクション内の全単語をリスト表示します。

```
if( wordCollection->getCount() > 0 )
{
   wordCollection->print();
   cout << "Total word count = " << wordCollection->getCount() <<
          endl;
   }
else
   cout << "No words in WordCollection!" << endl;
```

再度繰り返し処理関数について

print メンバ関数は，以下に示すように，*printWord* を指定して ***forEach*** 繰り返し処理関数を呼び出しています。

```
/* 繰り返し処理関数 */
static void printWord( void *w, void * )
{
   char *s = (char *)w;
   cout << s << endl;
}

void TWordCollection::print()
{

   forEach( &printWord, 0 );

}
```

forEach メンバ関数（もともと ***TNSCollection*** から継承！）は，コレクションをひと通り当たっていき，指定された関数に，その各要素を渡していきます。

要素の検索

ソート済みコレクションには（したがって文字列コレクションにも），ある特定のキーを持つ要素のインデックスを返す ***search*** メンバ関数が用意されています。でも，コレクションがソートされてい

ない場合，どのようにして要素を検索すればよいのでしょうか。あるいは，検索条件がキー自体ではない場合はどうでしょうか。通常は，***firstThat*** および ***lastThat*** を使用するというのが答となります。任意の条件でテストを行なうような ***Boolean*** 型関数を定義して，***firstThat*** を呼び出せばよいわけです。重複を許すソート済みコレクションでは，***search*** は一致するものが複数個ある場合には，その中の**最初の**ものを探し出してきます。したがって，プログラムによっては，その先を走査して重複するものがあるかどうかチェックする必要もでてきます。

多態的なコレクション

上では，コレクションに任意のデータ型を動的に登録できることを見てきました。また，コレクションのデータを効率よくアクセスするためのメンバ関数がいろいろとあることもわかりました。実際のところ，***TNSCollection*** には，18個を超えるメンバ関数が継承または定義されています。また，自分のプログラムでコレクションを使用してみればわかることですが，コレクションの処理速度についても目を見張ることと思います。コレクションは，いろいろと利用できるとともに，速度も速くなるように作られています。

しかし，コレクションの**真価**は，次の点にあります。すなわち，要素を多態的に扱うことができるという点です。つまり，さまざまな種類のオブジェクト型を登録することができるわけです。それもクラス階層中のどこに位置するものであってもかまいません。

これまで見てきたコレクションの例においては，コレクション内の要素は，すべて同じ型のものでした。文字列のリストでは，項目は，すべて文字列でした。顧客のコレクションの場合も同様でした。しかし，コレクションには，***TObject*** の下位型のオブジェクトであれば，**どんな**オブジェクトでも登録することができるのです。また，そのようないろいろなオブジェクトを自由に混在させることもできます。もちろん，それらのクラスには何か共通するものを持たせたいことになります。実際，抽象基本クラスを共有させたいことがよくあります。

ここでは，3種類の図形オブジェクトをコレクションに登録するプログラムを考えてみることにします。そのプログラムでは，***for-Each*** 繰り返し処理関数を使用して，コレクション内の各オブジェクトを1つずつ当たっていきながら，表示していきます。このスケルトンプログラムは，コレクション，図形オブジェクト，および多態性についてさらにいろいろと試してみるための格好のテンプレートとなっています。

☞ この例では，ボーランドのグラフィックスライブラリとBGIドライバを使用するので，プログラムに # include <graphics.h> という行を入れ，TV.LIBとGRAPHICS.LIBの両方にリンクさせてくだ

さい。プログラムを実行する場合には，BGI ドライバのあるディレクトリに移るか，***initGraph*** 呼び出しでドライバのディレクトリを指定するようにしてください（たとえば，A:¥BORLANDC¥BGI など)。

まず，抽象基本クラスの定義を行ないます。

これは，TVGUID20.CPP です。

```
class TGraphObject : public TObject
{

public:

   int x, y;
   TGraphObject();
// この例では，コンストラクタが x, y にランダムな値を代入する
   virtual void draw() = 0;
// 純粋仮想関数--導出クラスで定義する必要がある
};
```

以下宣言を見ればわかるように，導出された図形クラスそれぞれが，純粋仮想関数 ***draw*** を再定義することで，自分自身で初期化を行なったり，グラフィックス画面に自分自身を表示したりすることができます。そこで，*TGraphObject* の導出クラスとして，点，円，および矩形を定義します。

```
class TGraphPoint : public TGraphObject
{

public:

   TGraphPoint();
// この例では，コンストラクタが x, y にランダムな値を代入する
   virtual void draw();

};

class TGraphCircle : public TGraphObject
{

public:

   int radius;
   TGraphCircle();
// この例では，コンストラクタが x, y,
// および radius にランダムな値を代入する
   virtual void draw();

};

class TGraphRect : public TGraphObject
{

public:
```

```
   int width, height;
   TGraphRect();
// この例では，コンストラクタが x, y, w,
// および h にランダムな値を代入する
   virtual void draw();
};
```

この3つのクラスは，すべて *TGraphObject* から位置を表わす *x*, *y* データメンバを継承していますが，サイズは3つとも異なります。*TGraphCircle* には *radius* が追加され，*TGraphRect* には *width* と *height* が追加されています。以下が，コレクションを作成するプログラムです。

```
TNSCollection *list = new TNSCollection(10, 5);    // コレクションの作成

   for (int i = 1; i < 20; i++)
   {
      switch ( i % 3 )
      {

         case 0:
         {
             TGraphPoint *gp = new TGraphPoint;
             list->insert( gp );
             break;
         }
         case 1:
         {
             TGraphCircle *gc = new TGraphCircle;
             list->insert( gc );
             break;
         }
         case 2:
         {
             TGraphRect *gr = new TGraphRect;
             list->insert( gr );
             break;
         }
      }
   }
...
```

このプログラムでは，**for** ループで図形オブジェクトを20個 ***list*** コレクションに挿入しています。ここでは，***list*** の各オブジェクトが *TGraphObject* の導出クラスであるということがわかっているわけですが，挿入してしまえば，コレクション内のある要素が円なのか，点なのか，あるいは矩形なのかについては，わからなくなります。多態性のおかげで，それを知る必要はありません。各オブジェ

クトが，必要なデータとコード（*draw*）を内蔵しているわけですから。以下のように，単に繰り返し処理メンバ関数を使ってコレクションの各オブジェクトに1つ1つ当たっていきながら，それぞれのオブジェクトに自分自身を表示させるようにすればよいわけです。

```
void callDraw( void *p, void * )
{
   ((TGraphObject *)p)->draw();
// しかるべき draw メンバ関数を呼び出す
}

void drawAll( TNSCollection *c, void * )
{

   c->forEach( &callDraw, 0 );    // 各オブジェクトの表示
}
...

   drawAll( list, 0 );
...
```

関連性のある種類の異なる複数のオブジェクトを登録できるというコレクションの性質は，オブジェクト指向プログラミングの持つ強力な基礎を背景にしているものです。第8章では，この同じ多態性の原理が，ストリームにやはり有効に適用できることを示します。

コレクションとメモリ管理

コレクションオブジェクトは，そのコンストラクタの引数で設定した要素数の初期値から CONFIG.H で次のように定義されている ***maxCollectionSize*** で指定される最大値まで，動的に要素をふやすことができます。

```
const maxCollectionSize = (int)((65536uL - 16)/sizeof( void * ));
```

PC では要素へのポインタにメモリを 4 バイト使用するので，これでコレクションには最大 16,380 個の要素を持たせることができることになります。

動的データ構造を使用するライブラリの場合，エラー検出用としていくらかメモリを余分に用意しないかぎり，完全なものとすることはできません。コレクションを初期化する際に充分なメモリがない場合には，ヌルポインタが返されます。

コレクションオブジェクトに要素を 1 個追加する際にメモリが足りない場合には，静的メンバ関数 ***TNSCollection::error*** が呼び出され，実行時のヒープメモリエラーが発生します。***error*** は，以下のように定義されています。

```
void TNSCollection::error( ccIndex code, ccIndex )
{
   exit(212 - code);
}
```

したがって，***error*** は，デフォルト処理として，実行時エラー (212−code) を返します。たとえば，現在の最大値を超えるインデックスの位置にある要素を削除しようとした場合，***removeAt*** は ***error***(1, 0) を呼び出し，エラーコードを 211 にして終了します。***TNSCollection::error*** をオーバーライドすることによって，エラーを報告したり，エラーから回復するための独自の機能を用意することもできます。現在使用されてない 2 番目の引数を利用すれば，独自のエラールーチンに，補助データを渡すこともできます。

ヒープの利用可能性

Turbo Visionプログラムの場合，従来のプログラムの場合よりも，ユーザが制御を行なえる割合がかなり大きくなっていますので，ヒープが利用できるかどうかについては，とくに注意を要します。コレクションにオブジェクトを追加するという制御をユーザが行なう場合（たとえば，デスクトップに新たなウィンドウをオープンすることによって），それほど簡単には，ヒープエラーが発生しないようにはできません。コレクションを利用する際に自分自身でメモリチェックを行なうようにするか，あるいはプログラムをエラーからうまく回復させるような実行時エラーハンドラを設けるかして，実行時の致命的エラーがユーザに影響を及ぼさないように，策を講じる必要があります。

第8章
ストリーム入出力可能なオブジェクト

この章では，ストリームマネージャを用いてオブジェクトに**持続性**をもたらす方法について解説します。アプリケーションを起動してから終了するまでに，様々なオブジェクト（ウィンドウ，ダイアログボックス，コレクションなど）が作成されますが，その命はとても短いものです。オブジェクトはアプリケーションの進行に応じて，組み立てられ，表示され，アクセスされ，そして最後には消去されます。オブジェクトはそのスコープに出入りするごとに現われたり消えたりして，プログラムが終了する際に完全に消失します。ストリームマネージャを使えば，オブジェクトをメモリまたはファイルストリームにセーブすることで，そのオブジェクトを通常の寿命を超えた**持続的なものとする**ことができます。

持続的なオブジェクトには数えきれないほどの用途があります。たとえば，共有メモリにセーブした場合，オブジェクトを使ってプロセス間の通信を行なうことができます。また，モデムを介してオブジェクトを他のシステムに伝送することもできます。そして最も重要なことは，ファイルストリームを使ってオブジェクトをディスクに永久にセーブできるという点です。ディスクにセーブすれば，同じアプリケーションまたは別のアプリケーションを使って読み戻して復元することができます。Turbo Vision のオブジェクトはすべて，効率的で安全なストリーム入出力が可能です。ストリーム入出力する必要があると考えられるオブジェクト（グループ，ビュー，***TCollection*** の導出クラス，およびリソース）は，すべてストリーム入出力可能なクラスにしてあるので，通常のファイル I/O と同様に簡単にストリーム入出力することができます。

みなさんがストリーム入出力可能な独自のクラスを構築することも簡単に行なうことができます。クラスをストリーム入出力可能に

しても，オーバーヘッドはほとんど発生しません。その他，ストリーム入出力可能なクラスとするためには，***read***，***streamableName***，および ***write*** の 3 種類の仮想関数を ***TStreamable*** と呼ばれるクラスから継承する必要があります。ストリーム入出力可能なクラスは，すべて ***TStreamable*** から直接または間接に導出したものでなければなりません。***TView*** は ***TStreamable*** をすでにその基本クラスの 1 つとしているので，***TView*** から導出されたクラスはすべてストリーム入出力可能です。同様に，***TGroup*** の導出クラスもすべてストリーム入出力可能で，ユーザアプリケーションオブジェクトも例外ではありません。

ストリームオブジェクトは，***pstream*** とその導出クラスだけを使って作成されます。これらのクラスは，持続的なストリーム入出力のために特別に用意されているものですが，C++ 標準の **iostream** クラスにかなりよく似ているので，C++ のストリームをすでに熟知している方なら，新たに習得することはほとんどありません。この章では，ストリームはよく知らないという方のためにまず概要を簡単に説明し，基本的な用語を理解してからストリームの詳細な説明に入っていきます。

ストリームは無限

C および C++ には，I/O を行なうためのキーワードや定義済みの演算子はありません。どちらの言語も，標準ライブラリ (C では **stdio**, C++ では **iostream**)の関数を利用します。C++ のライブラリの場合には，オブジェクト指向プログラミングの能力を駆使することにより演算子をオーバーロードすることができたり，標準のデータ型かユーザ定義のデータ型かを問わずI/O の型保障を行なうようになっており，柔軟性，拡張性，および保全性を高めています。C++ では **stdio** の関数(**printf** など)を利用することもできますが，ほとんどの C++ プログラマはストリームのほうを利用しています。ところで，ストリームとはいったい何なのでしょうか。

ストリームとは，要素を無制限に(もちろん理論上ですが)並べ，いろいろなアクセス機能を備えた抽象データ型のことです。ストリームは，長さ(要素数)，現在位置(特定時点における一義的なアクセスポイント)，およびアクセスモード(入力には読み出し専用，出力には書き込み専用，入出力には読み出し/書き込み)を持ちます。読み出し(つまりデータの取り出し)は現在位置で行なわれますが(読み出しを行なうと現在位置は，通常，次の項目に進む)，書き込み(つまり挿入) はストリームの最後尾に項目を付加することで行なわれます。

従来のディスクファイルもストリームを採用していることで知られていますが，その概念をさらに拡張して，メモリに保存されているストリーム，キーボードから入力された文字のストリーム (標準入力 **cin** など) および画面に出力されたストリーム(標準出力，**cout**, **cerr** など)，そしてシリアルポートやその他のデバイスとの間でやりとりされるストリームも扱えるようにしました。このような考え方は，「すべてをファイルとして扱う」UNIX から派生しています。実際，オブジェクト指向の技法を使用すれば，どのようなデータソース(つまり生産側)であっても，取り出しメンバ関数を定義することで，入力ストリームとして扱うことができるようになります。同様に，ほとんどのデータシンク(つまり消費側)は，挿入メ

ンバ関数を定義することで，出力ストリームとして扱うことができるようになります。

ディスクファイルにリンクされるストリームは，普通，入力と出力の両方を行なうことができます。ストリームの各種クラスは，バッファ使用またはバッファ未使用，フォーマット指定またはフォーマット未指定，メモリ内またはファイル上，そして入力，出力，および入出力の，様々な組み合わせを表わします。Turbo Vision は **iostream** ライブラリを基礎としており，Turbo Vision で使われるさらに複雑なオブジェクトのストリーム入出力も行なうことができるようになっています。

オーバーロード演算子 << および >>

C++ のストリーム入出力が成功していることの大きな理由として，出力（挿入）には <<，入力（抽出）には >> というチェイン可能な演算子をオーバーロードすることができるということがあります。たとえば構文が複雑な **printf** などの **stdio** ライブラリ関数は，次のような簡潔な文に置き換えることができます。

```
cout << "Hello, World" << endl;
```

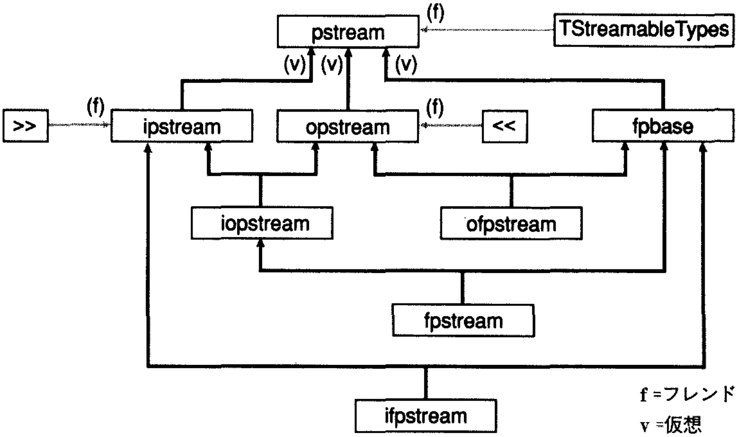
データ型とストリームクラスに対して，<< と >> を適切にオーバーロードすれば，その型のオブジェクトを適当なストリームオブジェクトに読み書きすることができるようになります。このようなオーバーロードは，char，short，int，long，char*，float，double，および void* など，C++ 標準のデータ型に「組み込まれて」います。また，ユーザが定義したクラス以外のデータ型にもオーバーロードを簡単に拡張することができます。クラスオブジェクトに対して同様のオーバーロードを行なうことは少し難しくなりますが，Turbo Vision ではすでにオーバーロードしてあるので，内部的な詳細を知らなくても，次のような文を使って，オブジェクトを読み書きすることができます。

```
os << aTVObject << anotherTVObject;
is >> yetAnotherTVObject;
```

ストリーム入出力可能なクラスを独自に作成および登録する場合には，この章で後述する，いくつかの単純な予備的作業を行なう必要があります。上の例では，*os* は ***opstream*** のオブジェクト（またはその導出クラスのオブジェクト），そして *is* は ***ipstream*** のオブジェクト（またはその導出クラスのオブジェクト）です。これらの 2 つのクラスは，それぞれ ***pstream*** の導出クラスで，持続的オブジェクトを作成するための基本ストリームクラスです（名前に付いている *p* は，「持続的」を意味する英語 persistent の頭文字)。これは，C++ 標準のクラス階層において **istream** と **ostream** が ***ios*** から導出されるのと同様です。

また，*ipstream* と *opstream* を結び付けている，*iopstream* というクラスもあります。さらに，*ifpstream*, *ofpstream*, および *fpstream* といったファイルストリームの導出クラスもあります。これらは C++ 標準の *ifstream*, *ofstream*, および *fstream* の各クラスに相当し，同じようなデータメンバとメンバ関数を持っています。**図 8.1** は，*pstream* クラスの階層を図示したものです。C++ の慣例に従い，矢印は基本クラスに向かっています。

図 8.1
pstream の階層



TView オブジェクトとオブジェクトポインタを *opstream* に書き込むために使用される，オーバーロード演算子 << は，VIEWS.H で次のように定義されています。

```
inline opstream& operator << ( opstream& os, TView& cl );
inline opstream& operator << ( opstream& os, TView* cl );
```

最初の演算子においては，*TView* オブジェクト *cl* が *opstream* オブジェクト *os* に書き込まれます。この演算子は同じ出力ストリームを返すので，C++ 標準の **ostream** クラスでよく使用される，<< 演算のチェインが可能です。2 番目の演算子では，*TView* オブジェクトへのポインタに対して同じことを行ないます。単に目的とするオブジェクトを型キャストして，基本クラスで定義されている << 演算子を呼び出すようになっています。この場合，*TView* の基本クラスは *TStreamable* です。

Turbo Vision では，標準のストリーム入出力可能なクラスすべ

てについて同じようにオーバーロード演算子 << および >> が用意されています。これら演算子の処理内容は通常，インラインで定義され，上と同様の形式に従います。プログラマ独自のストリーム入出力可能なクラスに << および >> をオーバーロードしたい場合，普通，同様のプログラムを書く必要があります。このように言うと，説明が循環しているように思われるかもしれません。ストリーム入出力可能なクラスは，いったいどのようなものなのでしょうか。以下をお読みください。

ストリーム入出力可能なクラスと TStreamable

ストリーム入出力可能なクラスとは，Turbo Vision のストリームマネージャに備わっているツールを用いて，そのオブジェクトを持続的なストリームに読み書きできるクラスのことです。ストリーム入出力可能なクラスは，適当な入出力演算子（通常はオーバーロードされた << および >> ですが，独自の演算子を定義してもかまいません）以外にも，その階層のどこかで ***TStreamable*** を基本クラスとする必要があります。Turbo Vision のクラスツリー（99 ページ）を見ると，***TView*** は ***TObject*** および ***TStreamable*** を多重継承していることがわかります。したがって ***TView*** から導出されたクラスは，すべて ***TStreamable*** の魔術を継承しています。標準ビュークラスも，すべてオーバーロード演算子 << および >> を持っていますのでストリーム入出力可能です。

ビューでないクラスとはどのようなものでしょうか。第 7 章で解説したとおり，***TCollection*** は ***TStreamable*** を基本クラスとしています（***TNSCollection*** 基本クラスもです）。さらに，***TCollection*** オブジェクトにはオーバーロード演算子 << および >> が定義されているので，***TCollection*** から導出されたすべてのコレクションクラスは，ストリーム入出力可能です。

ストリームマネージャ

ストリームとの間で複雑なオブジェクトをセーブしたり読み出したりするためには，舞台裏である程度の管理を行なう必要があります。これを行なうのが，ストリームマネージャの仕事です。ストリームに単純なデータオブジェクトを書き込む場合，そのオブジェクトはあまり複雑な作業を行なわなくても，単純なバイナリイメージとして保存したり読み出したりすることができます。ところがクラスオブジェクトには，他の複雑なオブジェクトを指すポインタや，vtable (仮想メンバ関数のテーブル) を指すポインタなど，いろいろなデータ型をいくらでも入れることが可能です。

このような様々な状況に対処するため，ストリームマネージャは ***TStreamableTypes***，***TStreamable***，および ***TStreamableClass*** を使って，ストリーム入出力可能なクラスのデータベースを維持しています。オーバーヘッドは，ストリーム入出力可能なクラスあたり (インスタンスあたりではありません) 16 バイト弱です。これらのクラスを組み合わせることにより，各オブジェクトに対する基本的な登録，***read***，***write***，および ***build*** 関数が実現されます。ストリームマネージャにクラスを登録するのは，データベースに正しく登録することで，ストリームマネージャがそのクラスをストリーム入出力可能なクラスとして認識できるようにするためです。

詳しくは，第 13 章の TPWrittenObj および TPReadObj のエントリを参照してください。

さらに，ストリームとの間で読み書きしている場合，ストリームマネージャは読み書きされたすべてのオブジェクトのデータベースを維持しなくてはなりません。これについてはあまり深入りせずに (バックグラウンドで行なわれる処理なので)，ポインタを使ってストリームにオブジェクトを書き込む場合の問題点を考えてみましょう。*ptr1* と *ptr2* が同じオブジェクトを指し，**ptr1* と **ptr2* の両方をストリームに書き込むとします。後でこれらのオブジェクトをストリームから読み出す際，異なるポインタを持つ同一のオブジェクトが 2 つ読み出されることになり，混乱をきたします。ストリームマネージャのデータベースは書き込まれたオブジェクトを保存するとともに，**ptr1* を 1 つだけストリームに書き込むことでこのような問題が起こらないようにします。同様に，オブジェクトをスト

リームから **nptr1* および **nptr2* の両方に読み込む際，オブジェクトは 1 つだけ作成され，*nptr1* と *nptr2* は両方ともそのオブジェクトを指すようにされます。

またストリームマネージャは，適当な ***read*** および ***write*** 関数を用意することで，ストリームオブジェクトに対する密接な処理を行ないます。***TStreamable*** では，純粋仮想関数である ***read*** および ***write***（リーダおよびライタとして知られています）を用意しています。これらの関数は，***TStreamable*** から導出されたストリーム入出力可能な各クラスにおいて再定義する必要があります。

```
class TStreamable
...
protected:

    virtual void *read( ipstream& ) = 0;
    virtual void write( opstream& ) = 0;
```

ライタ(書き込み関数)の仕事は，オブジェクト中の必要なデータメンバをすべてストリームに書き込むことです。ストリーム入出力可能なクラスは，それぞれライタを用意する必要がありますが，ライタの処理内容は，その基本クラスのライタを呼び出すことで簡単に定義できます。以下は，***TView::write*** および ***TGroup::write*** の例です。

```
void TView::write( opstream& os )
{
    ushort saveState =
        state & ~( sfActive | sfSelected | sfFocused | sfExposed );

    os << origin << size << cursor
       << growMode << dragMode << helpCtx
       << saveState << options << eventMask;
}
...
void TGroup::write( opstream& os )
{
    ushort index;

    TView::write( os );
    TGroup *ownerSave = owner;
    owner = this;
    int count = indexOf( last );
    os << count;
    forEach( doPut, &os );
    if (current == 0)
       index = 0;
```

```
    else
       index = indexOf(current);
    os << index;
    owner = ownerSave;
}
```

read 関数はライタと対応した処理を行ないます。ストリーム入出力可能なクラスとするためには，***TStreamable*** が用意している純粋仮想関数 ***read*** を再定義する必要があります。通常，基本クラスのリーダ（読み出し関数）を拡張して，追加されたデータメンバも扱えるように再定義します。

読み出しを行なうときに特殊なケースが発生します。ストリームからオブジェクトをしかるべき型のオブジェクトに読み出す場合，***read*** 関数は単に必要なデータと vtable のポインタを転送するだけです。しかしながら，読み出し先としてそのようなオブジェクトがない場合には，まずオブジェクトを1つ作成する必要があります。すなわちストリームが初期化できるようにした一種のスケルトンオブジェクトを作成する必要があります。これがビルダ（組み立て関数）の役割です。ビルダはメモリを割り当て，vtable のポインタを設定します。ストリーム入出力可能なクラスで，読み出しに関するこのような処理を行なえるようにするためには，ビルダと ***build*** コンストラクタを用意する必要があります。

Turbo Vision 標準のストリーム入出力可能なクラスには，リーダ，ライタ，ビルダ，および ***build*** コンストラクタがあらかじめ定義されています。ストリーム入出力可能なクラスを独自に作成する場合には，これらの関数を用意する必要があります。

ストリーム入出力可能なクラスのコンストラクタ

前述したように，ストリーム入出力可能なクラスの自動または静的オブジェクトがストリーム読み出し処理の対象となる場合には，特別なコンストラクタが必要です。そのコンストラクタは，引数として ***streamableInit*** だけを取ります。標準のストリーム入出力可能なクラスは，すべてこのようなコンストラクタを用意しています。通常，以下のように protected にされます。

```
class TMyStreamable : public virtual TBase, public TStreamable
{
...
protected:

   TMyStreamable( StreamableInit s);
...
};
```

StreamableInit というデータ型は，***streamableInit*** だけをメンバにする enum です。以下のコンストラクタを使ってオブジェクトを作成する場合，標準のコンストラクタの場合は異なり，中に入っているポインタのコンストラクタは呼び出されません。

```
TMyStreamable str( streamableInit );
```

この結果，メモリの管理が単純化されます。ストリームからオブジェクト *str* へとオブジェクトを読み出す場合でも，そのオブジェクトのリーダは *str* から不要なデータを解放しなくてもすむのです。

```
  TMyStreamable str( streamableInit );
// "空の" str を作成
...
  ifpstream ifps("str.sav"); // ファイルストリームを入力用にオープン
  ifps >> str;               // オブジェクトを str に読み込む
...
```

注意：*str* など，***streamableInit*** コンストラクタを用いて作成したオブジェクトは，ストリーム読み出し処理によって「初期化」されるまでは安全に使用することはできません。また，列挙型およびそのメンバの名前に特別の意味はなく，このコンストラクタを他のコンストラクタから区別するために引数として一義的なデータ型引

数を用意しているにすぎません。

ストリーム入出力可能なクラスの ***build*** メンバ関数は，次のように定義されています。

```
TStreamable *TMyStreamableClass::build()
{
   return new TMyStreamableClass( streamableInit );
}

TMyStreamableClass:: TMyStreamableClass( StreamableInit s ) :
   TBaseClass( streamableInit )
{
}
```

つまり ***build*** コンストラクタは，単にその基本クラスのコンストラクタを呼び出していっているにすぎないということです。

ストリーム入出力可能なクラスの名前

ストリーム入出力可能なクラスとするためには，***Tstreamable*** から継承したプライベートの仮想関数 ***streamableName*** をオーバーライドする必要があります。この関数は，そのクラスを表わすヌルで終了する一義的な文字列の名前を返さなければなりません。

```
class TMyStreamable : public virtual TBase, public TStreamable
{
...
private:

    virtual const char *streamableName() const;
        { return "TMyStreamable"; }
```

Turbo Vision 標準のすべてのストリーム入出力可能なクラスでは，この作業もすでに行なってあります。しかし，独自のストリーム入出力可能なクラスを自定義する場合には，独自のオーバーライドを行なう必要があります。通常は，上のようにして ***streamableName*** にクラス名を返させるようにします。ストリームマネージャは，この一義的な名前を使用して，クラスの各種データベースを管理します。

ストリームマネージャの使用

ストリームマネージャの使用は,以下の 3 つのステップからなります。

1. ストリームマネージャのプログラムにリンクする。
2. 適切なストリームオブジェクトを作成する。
3. ストリームオブジェクトを使用する。

では,各ステップを詳しくみてみましょう。

ストリームマネージャのプログラムへのリンク

ストリーム入出力可能なクラスはすべて, ***read***, ***write***, ***build*** という 3 種類の関数を定義することで,そのクラスのオブジェクトを扱えるようにしなくてはなりません。これらの関数はストリームマネージャに知らせる必要があり,また,ストリームマネージャを使うアプリケーションにリンクさせなくてはなりません。ここで述べるリンクとは,クラス名をストリームマネージャに**登録すること**であり,これはマクロ_ _***link*** の呼び出しを 1 回行なうだけですみます。このマクロでは *RClassName* を使う必要があります。*ClassName* は,ストリーム入出力可能なクラスの名前から頭文字の *T* を除いたものです。たとえば, ***TChDirDialog*** というクラスをストリーム入出力したい場合,プログラムに次のような文を追加する必要があります。

```
_ _link(RChDirDialog);
```

ストリームオブジェクトの作成と使用

ストリームオブジェクト ***ipstream*** または ***opstream*** の作成は, ***iostream*** の各クラスの場合と同様,適当な引数を用いて宣言する

だけで済みます。たとえば DLG.SAV というファイルストリームにダイアログをセーブするには，次のようにして ***ofpstream*** オブジェクトを宣言するだけで済みます。

各種のストリームコンストラクタおよびその引数については第13章を参照してください。

```
TChDirDialog cdlg;
...
// ここでダイアログを作成

ofpstream of( "dlg.sav" );
// 出力ファイルストリームをオープン

of << cdlg;
// ファイルストリームにダイアログオブジェクトを書き込む
```

ここで，コンストラクタ

```
ofpstream(  const char *filename, int mode = ios::out,
            int prot = filebuf::openprot)
```

が，引数にデフォルト値をとって呼び出されています。

通常読み出し処理は以下のようなものとなります。

```
TChDirDialog cdlg (streamableInit );
// ビルドコンストラクタを呼び出して，オブジェクトの骨格を作る

ifpstream ifps( "dlg.sav" );
// 入力ファイルストリームをオープン

ifps >> cdlg;
// ファイルストリームから cdlg にダイアログオブジェクトを読み込む
```

ストリーム上のコレクション

第 7 章『**コレクション**』で，種類は違うが関連のある複数のオブジェクトをコレクションにできることを説明しましたが，この多態性は，そのままストリームにも適用することができます。ストリームを使って，コレクションをまるごとディスクに保存し，あとで読み出したり，別のプログラムから読み出したりすることができます。後戻りして，232 ページの TVGUID20.CPP を眺めてみてください。このプログラムで，コレクションをストリームに乗せるためには，何を付け加えればよいのでしょうか？

答えは驚くほど単純です。まず，基本クラスのオブジェクト ***TGraphObject*** に対して，そのデータ（x と y）をストリームに保存する方法を「教えます」。これは，***write*** メンバ関数で行ないます。そして同様に，***TGraphObject*** の下位オブジェクトのうち，データメンバを追加するものについても，新たに ***write*** メンバ関数を定義します（***TGraphCircle*** が ***radius*** を追加し，***TGraphRec*** が ***width*** と ***height*** を追加します）。

その他 ***readItem*** および ***writeItem*** も用意する必要があります。***readItem*** は型を保障しつつ，***ipstream*** から要素を読み戻し，***writeItem*** は要素を ***opstream*** に書き込みます。

また，実際に保存したい各クラスを登録することも忘れないでください。これで終わりです。あとは通常のファイル I/O と同じようにしてストリーム変数を宣言し，新規ストリームを作成し，コレクションをまるごとストリームに書き込みます（文 1 個で済む）。そしてストリームをクローズすればよいわけです。

write 関数の追加

以下が ***write*** メンバ関数です。***TGraphPoint*** は，***TGraphObject*** から継承するデータメンバ以外何もデータメンバを追加しないので，***write*** メンバ関数は必要ありません。

```
class TGraphObject : public TObject
{

public:
    ...
    virtual void write( opstream& os);
    ...
};

class TGraphCircle : public TGraphObject
{

public:

    int radius;
    ...
    virtual void write( opstream& os);
};

class TGraphRect : public TGraphObject
{

public:
    ...
    int width, height;
    ...
    virtual void write( opstream& os);
};
```

write の処理内容の定義は簡単明瞭です。どのオブジェクトの場合でも，まず継承した ***write*** メンバ関数を呼び出し，継承したデータを全部保存します。それからそのオブジェクトで追加されたデータをストリームの ***write*** メンバ関数を呼び出して書き込みます。

これは TVGUID21.CPP です。

```
void TGraphObject::write( opstream& os)
{
  os << x << y;
}

void TGraphCircle::write( opstream& os)
{
  TGraphObject::write( os );
  os << radius;
}

void TGraphRect::write( opstream& os)
{
  TGraphObject::write( os );
  os << width << height;
```

TVGUID21.CPP を詳しく見ると，独自の ***read***, ***write***, および

build 関数を作成する方法や，ストリーム入出力可能なクラスを登録する方法がわかるはずです。

デスクトップのセーブと復元

ストリームにセーブしたいオブジェクトがデスクトップである場合，デスクトップが所有するものすべて，すなわちその時点でデスクトップ上に作成されているすべてのビューを含むデスクトップ環境全体は，そのデスクトップによってセーブされます。

ユーザがデスクトップをセーブできるようにするには，セーブされるすべてのビューが適切な ***write*** および ***read*** メンバ関数を備え，かつ，すべてのビューが登録済みのものとなっているようにしなくてはなりません。デスクトップ上に，いつどういったものが作成されるかは，たいていユーザによって決定されるからです。

さらに一歩進んで，アプリケーションをまるごとセーブしたり，復元することもできます。***TApplication*** オブジェクトは，自分自身をセーブしたり，復元することができます。

第9章
リソース

リソースファイルに保存できるものは，何でもリソースだと皮肉な定義をしている人もいますが，もっとわかりやすく言えば，そのような人達は，リソースファイルはリソースを保存するための場所だと定義しているわけです。確かに，コンピュータの世界では，**リソース**という言葉は，ほとんど何でも意味するような曖昧な言葉になっています。通常このマニュアルでは，リソースは，メニューやキャプションやダイアログボックスなどのことを意味します。

Turbo Vision では，リソースファイルは渡されてきたオブジェクトをセーブし，そのオブジェクトを名前で読み出せるようにしたTurbo Vision オブジェクトです。したがってアプリケーションは，自分自身が使うオブジェクトを，コンストラクタを呼び出さなくてもリソースから選択的にロードすることができます。また，オブジェクトの作成と初期化をアプリケーションにやらせなくても，別個のプログラムですべてのオブジェクトを作成し，作成したオブジェクトはリソースにセーブするようにすることができます。リソースは，アプリケーションの .EXE コードの一部とすることや，別個のファイルにすることもできます。

Turbo Vision には，***TResourseCollection*** および ***TResourceFile*** というクラスがあります。これらのクラスを利用すれば，第8章で解説した従来のストリームによる場合よりも多様な方法で，ストリームとの間で要素(通常，上記以外のクラスのオブジェクト)の読み書きを行なうことができます。そのような多様な機能を持たせることができるのは，***TResourceFile*** オブジェクト内の要素に名前でインデックスがつけられるからです。名前キー（普通の文字列）は，要素をセーブする際に指定します。その要素の読み出しは，同じプログラムからでも別のプログラムからでも行なうことができ

ますが，その際にはこのキーを指定します。アプリケーション側では，その要素がストリーム内のどこに格納されているか，またその要素のサイズはどれだけであるかといったことを管理する必要はありません。したがって Turbo Vision のリソースファイルは，インデックス付きのランダムファイルによく似ていると言えます。

またリソースファイルには .EXE ファイルに付加できるという利点もあります。つまり，アプリケーションに既成のオブジェクトを取り込むことができるので，プログラムを単純化できるということになります。リソースが .EXE の一部として作られるか，外部ファイルとして作られるかにかかわらず，アプリケーションの用途を大幅に広げることができるようになります。見出しやヘルプスクリーンに使う言葉など，多くのものをプログラムコードから分離することができます。リソースエディタを使えば，この機能の多様性をさらに高めることができます。

リソースの要素は，***TResourceFile*** が所有する ***fpstream*** オブジェクトに格納されますが，効率をよくするために，文字列キーは ***TResourceCollection*** オブジェクトでソートして管理されます。***TResourceCollection*** には，ストリーム上のリソースに対応させた位置およびサイズのフィールドが設けられています。

TResourceCollection は，***TSortedCollection*** の導出クラスである ***TStringCollection*** の単純な導出クラスです。***TResourceCollection*** は ***TStringCollection*** にメンバを追加して，リソースインデックスの処理を行なえるようにしています。アプリケーション側でリソースファイルにリンクされる ***TResourceCollection*** オブジェクトに直接関与することはほとんどないでしょう。***TResourceFile*** コンストラクタはリソースコレクションを作成し，***TResourceFile*** の各メンバ関数がそのコレクションとのやりとりや管理を行ないます。アプリケーション側では，適当なファイルストリームを作成する必要があります。そのためには，ファイル名，アクセスモード（ ***ios::in, ios::out*** など ），保護モードなどを引数で適当に指定して ***fpstream*** コンストラクタの中のどれかを呼び出します。

リソースの仕組みは，きわめて簡単なものです。リソースファイルは，ランダムアクセスのストリームと同様の機能を果たします。各リソースに対応した一義的な文字列を**キー**にしてオブジェクトを

アクセスするわけです。***TNSSortedCollection***には，***TSorted-Collection***から***duplicates***データメンバを継承しています。***duplicates***は，デフォルトで*False*に設定されます。したがって，キーの重複は許されないようになっています。リソースを利用する通常のアプリケーションにおいては，このフィールドを*False*にしたままにしてキーの重複を避けるようにしてもよいでしょう。デフォルトにした場合には，既存のキーを使ってリソースをセーブすると，新しいリソースがそれまでのリソースのかわりに格納されることになります。

Turbo Visionの他の仕掛けとは違って，リソースについては，何か変更しなければならないということはないでしょう。リソースは，そのままでも強力な機能を備え，いろいろな用途に使うことができます。

リソースが必要なわけ

リソースファイルには，いろいろな利益があります。

1. まず，リソースを利用することで，アプリケーションをカスタマイズする際にコードに手を加える必要がなくなります。たとえば，ダイアログボックスのテキスト，メニューアイテムのラベル，ビューの色などは，どれも，リソースで変更することができます。アプリケーションの内部に立ち入らなくても，アプリケーションの形態を変更できるわけです。
2. リソースを使うことで，一般にコードサイズを小さくすることができます。アプリケーションのメインルーチンとは別個のプログラムで，複雑なオブジェクトを作成したり，そのオブジェクトをリソースファイルに保存するようにすることができ，メインルーチンでそれを簡単に読み出すことができるようにすることができます。
3. リソースを利用すれば，アプリケーションを国語別に対応させる手間も簡単になります。アプリケーションでは，オブジェクトを名前でロードすることができますが，オブジェクトが表示する国語は，各オブジェクトで管理する必要があります。
4. アプリケーションに，いくつかのバージョンを設け，それぞれ機能が異なるようにしたい場合，たとえば，メニューを2組作り，一方のメニューでは全部の機能が使えるようにし，もう一方のメニューでは使用できる機能を制限するというようなことも行なうことができます。このようにすれば，プログラムを書き直す必要はなく，したがって必要な部分を間違って削除してしまうということもなくなります。また，プログラムを改訂して全機能をサポートするようにしたい場合も，単に新しいリソースを追加するだけでよく，プログラム全体を元通りに書き直す必要はありません。つまりリソースは，プログラムで使用するオブジェクトの表現部分を隔離し，表現を簡単に変更できるようにするためのものです。

リソースの内部

みなさんは，リソースの詳細に立ち入るよりも，リソースがストリームとコレクションを利用しているのであれば，まず，ストリームとコレクションについて充分理解しておきたいと思われるかもしれません。リソースを**利用する**だけであれば，リソースの仕組みがどうなっているかについては知る必要がないのですが，システムをもうひとひねりしようと思っているのであれば，以下に述べる詳細を知っておくと役立つでしょう。

TResourceFile オブジェクトは，ソート済み文字列コレクション（***TResourceCollection*** オブジェクト）と入出力ファイルストリーム（***fpstream*** オブジェクト）を所有しています。このオブジェクトの所有関係は，***TResourceFile*** の 2 つのデータメンバによって決定されます。

```
TResourceCollection *index;
// コレクションへのポインタ

fpstream *stream;
// ストリームへのポインタ
```

コレクション内の文字列は，ストリームのオブジェクトに対するキーになっています。　コレクションは，そのデータメンバ ***limit***, ***delta*** にしたがって，動的に拡張することができることを思い出してください。***TResourceCollection*** の各要素は ***TResourceItem*** 構造体のオブジェクトです。

```
struct TResourceItem
{

   long pos;
   long size;
   char *key;

};
```

key メンバは，リンクしているファイルストリーム内の基本位置 ***pos***（ヘッダ情報を無視しての位置）のところにあるオブジェクトに対する文字列キーです。***size*** メンバは，インデックスがつけられた

オブジェクトの長さを指定します。

文字列コレクションは，***TResourceFile*** コンストラクタによって作成され，***TResourceFile*** のメンバである ***get*** や ***put*** などからアクセスされたり管理されたりします。ただし，リソースファイルを作成する**前に**，ファイルストリームをオープン（作成，初期化）しておく必要があります。実際問題として，***TResourceFile*** コンストラクタを呼び出すときには，引数に，すでに存在している ***fpstream*** ポインタを指定する必要があります。

```
TResourceFile::TResourceFile( fpstream *aStream );
// 指定した aStream をもつリソースファイルを作成します
```

リソースの作成

リソースファイルの作成と利用は，4段階の処理からなります。すなわち，ストリームをオープンし，そのストリームでリソースファイルを初期化し，1個以上のオブジェクトをキーを付けて登録し，そして終了したらファイルストリームをクローズする，という4段階です。

実際に稼動する Turbo Vision のリソースの活用例が，デモ用のソースコード GENFORM にあります。

以下のプログラムは，リソースとして 'Waldo' というキーを持つステータスラインを1個だけ格納する MY.REZ という簡単なリソースファイルを作成するものです。***TResourceFile::put*** は，引数を2つとります。格納されるストリーム入出力可能な要素へのポインタ TStreamable* と const char* キーの2つです。適当な .H ファイルをインクルードし，ストリーム入出力可能なクラスを登録するために，#include<tv.h>の前に，いくつか #defile を行なう必要があります。

```
/* 実際のアプリケーションでは、define 文と includes 文は、
別々のヘッダファイルに持たせたりします */

#define Uses_TRect
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
/* 最後の #defines 2つは冗長ですが、害はありません--
以下の Uses_TResourceCollection にある注を参照してください */

#define Uses_TResourceFile
#define Uses_TResourceCollection
/* 実際には #define Uses_TResourceCollection は必要ありませんが、
何も害はありません：TV.H は、Uses_TResourceFile の定義から、
RESOURCE.H がインクルードされることを知っています */

#define Uses_fpstream
...
// 各クラスで使用される #define 文をさらに追加します


...
// ここで、使用するストリーム可能なクラスを登録します
_ _link(RStatusLine);
_ _link(RResourceCollection);
/*
TResourceFile は、ストリーム不可 (_not_ streamable!) です。
リソースコレクションだけがストリームされます。
*/
...
// その他のストリーム可能なクラスを登録するために
// #define 文をさらに追加します。
...
#include <tv.h>
// これによって、必要な .H ファイルだけが取り込まれます。

#include <iostream.h>
// 必要に応じて、さらに標準的なインクルードを行ないます。


...

const char rezFileName[] = "MY.REZ";

cout << "Creating " << rezFileName << endl;

fpstream *ifps;
ifps = new fpstream( rezFileName, ios::out|ios::binary );
if ( !ifps->good() )
{
    cerr << rezFileName << ": init failed..." << endl;
    exit(1);
}
// TResource コンストラクタを呼び出す前に、ストリームが OK か
// どうかチェックします。fpstream を出力／バイナリモードで
// オープンします。第3引数 prot のデフォルトは
// filebuf::openprot です。エラーが発生した場合には、
// メッセージボックスを使って、正しい使用法を表示します。
```

```
// デモ用ソースコードとして納めてある LISTDLG.CPP を
// 参照してください。

TResourceFile *myRez;
myRez = new TResourceFile( ifps );
if ( !myRez )
{
    cerr << "Resource file init failed...";
    exit(1);
}

// この時点で myRez.stream の ifps への設定が完了し、
// 格納できる状態になっています。

// つぎにアプリケーションで使用するステータスラインを作成します。

TRect r( 0, 24, 80, 25 );
TStatusLine *sl;
sl = new TStatusLine( r
       *new TStatusDef( 0, 0xFFFF ) +
         *new TStatusItem( "~GRPH-X~ Exit", kbAltX, cmQuit ) +
         *new TStatusItem( "~GRPH-F3~ Close", kbAltF3, cmClose )
         );
if (!sl)
{
    cerr << "StatusLine init failed...";
    exit(1);
}

myRez->put( sl, "Waldo");
/* ステータスラインをリソースに "Waldo" というキーを付けてセーブ
します。より完全なプログラムとする場合には、まず myRez にすでに
そのキーがあるかどうかをチェックして、すでにあるリソースを上書き
してしまう可能性があることをユーザに警告する必要があるでしょう。*/

destroy sl;
// 必要なくなったので。

destroy myRez;
// リソースを MY.REZ にセーブしたので、
// myRez オブジェクトを片づけます。
...
```

リソースの読み出し

リソースファイルからのリソースの読み出しは，セーブするのと同じように簡単に行なうことができます。引数に目的のリソース要素のキーを指定して ***TResource::get*** を呼び出せばよいのです。***get*** は，その要素への汎用の void*ポインタを返すので，返されたオブジェクトを処理する前に型キャストを行なう必要があります。

上のプログラムで作成したステータスラインソースを，アプリケーションで読み出して使用するには以下のようにします。考え方としては，通常なら ***TProgInit*** コンストラクタを介して ***initStatusLine*** を呼び出すところを，MY.REZ から「作成済みの」ステータスラインを読み出してくるコードに置き換えればよいわけです。

ここでは，#define 文と#include 文は省いて，単純化しています。

```
const char rezFileName[] = "MY.REZ";

class TMyApp: public TApplication
{

public:

   TMyApp();
   static TStatusLine *initStatusLine( TRect );
   ...
};
// この例では、メニューバーとデスクトップは省略しています。

TMyApp::TMyApp() : TProgInit( &TMyApp::initStatusLine )
{
}

TStatusLine *TMyApp::initStatusLine( TRect )
{

   fpstream *ofps;
   ofps = new fpstream( rezFileName, ios::in|ios::binary );
   if ( !ofps->good() )
       messageBox( "Stream open error", mfError|mfOKButton );
// TResource コンストラクタを呼び出す前に、ofps が非 0 かどうかをチェックします。
// fpstream を入力／バイナリモードでオープンします。第3引数
// prot のデフォルトは filebuf::openprot です。

   TResourceFile *myRez;
   myRez = new TResourceFile( ofps );
   if ( !myRez )
      messageBox( "Resource file error", mfError | mfOKButton);
```

```
    else
       return (TStatusLine *) myRez->get( "Waldo" );
}
...

int main()
{

    TMyApp waldoApp;
    waldoApp.run;
    return 0;

}
```

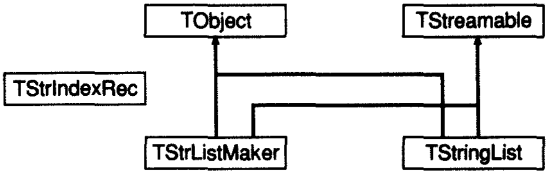
リソースからオブジェクトを読み出す場合，0ポインタが渡されてくる可能性もあるということを知っておく必要があります。インデックスの名前が間違っていた場合（すなわち，そのようなキーのリソースがそのファイルになかった場合），***get*** は0ポインタを返してきます。しかし，リソースコードをデバッグすれば，これも解決できます。

リソースから同じオブジェクトを何度でも読み出してくることもできます。上のステータスラインの例ではそのようなことをする必要はないでしょうが，たとえばダイアログボックスの場合，アプリケーションを実行している最中にユーザが何度も読み出させる可能性があります。リソースは，要求があれば，同じオブジェクトを何度でも読み出してきます。

文字列リスト

リソースの標準的な機能以外にも，文字列リストを処理するための特殊なオブジェクトも2個用意されています。**図 9.1** は，これらのオブジェクトが全体図のなかでどのような位置にあるかを示しています（C++の慣例にしたがって矢印で**指される**側が基本クラスです）。

図 9.1
文字列リストの階層



文字列リストは，その他のリソースに比べて，機能の多様性という点では劣りますが，設計の意図どおりに利用すれば高速で便利だという特長を備えています。

文字列リストは，単純化されたリソースであり，リソース化された文字列を，キー文字列ではなく数値（unsigned short）でアクセスできるようにしたものです。カスタマイズや多国語化を簡単に行なえるように，文字列をリソースファイルに登録できるようにしています。***TStringList*** と ***TStringCollection*** を混同しないようにしてください。文字列リストは ***ifpstream*** とリンクするリソースですが，文字列コレクションはストリームを持たない単に文字列をソートしたコレクションにすぎません。

たとえばボーランドのプログラマーズプラットホーム（総合環境）では，エラーメッセージは，すべて文字列リストに登録されています。したがってプログラムからは，番号でエラーメッセージを取り出してくることができます。また，国によってバージョンが違うときには，リソースの文字列を変えればよいようになっています。文字列リストと通常のリソースがどのように違うかは，リソース化された要素にインデックスをつけるときに使用される以下の2つのクラス ***TStrIndexRec***, ***TResourceItem*** を比較することでわかります。

```
struct TResourceItem
{

   long pos;
   long size;
   char *key;

};

class TStrIndexRec
{

public:

   ushort key;
   ushort count;
   ushort offset;

};
```

ビルダコンストラクタについては，第8章と第13章のTStreamableの項を参照してください。

TStringList では，コンストラクタとして引数に ***streamableInit*** を1つとるビルダしか定義されていません。というのは，文字列リストはリソースファイルにしかなく，リソースファイルからアクセスすることはできても，それ以外の場所で直接作成することはできないからです。文字列リストは基本的に読み出し専用のリソースなので，***get*** メンバ関数はありますが ***put*** はありません。

文字列リストの作成

TStringList は，文字列をアクセスするために使用されるストリーム入出力可能なクラスです。TStrListMaker が文字列リストを作成します。

TStringList の他にも，***TStrListMaker*** を使用する必要があります。***TStrListMaker*** は，***TStringList*** で使用するリソースファイルに文字列リストを作成します。文字列リストがリードオンリーなのに対して，文字列リストメーカはライトオンリーです。したがって ***TStrListMaker*** には，***put*** はありますが ***get*** はありません。文字列リストメーカで行なえることは，文字列リストを初期化し，それに文字列を書き込み，そしてできあがったリストをストリームに格納するということだけです。以下のコードは，文字列リストの作成と使用法を示したものです。

```
#define Uses_TStringList
#define Uses_TStrListMaker
...
#include <tv.h>
_ _link(RStringList);
...
TStrListMaker myStrListMaker( strSize, indexSize );
// 要素のサイズが strSize の文字列の配列を作成します。
// TStrIndexRec 型を要素とする indexSize の配列オブジェクトを作成します。
// 各要素は、データメンバとして key、count、および offset を持ちます。
// データメンバ index には、このTStrIndexRec オブジェクトの配列への
// ポインタが設定されます。

myStrListMaker.put( 1, "Pass, Do not GO!" );
myStrListMaker.put( 2, "Collect $500!" );
...
opfstream ops( "STR.LST" );
ops << myStrListMaker;
...
TStringList *myStringList;
// indexSize、index、および basePos に 0 を設定します。

char legend[maxLeg];

ipfstream ifs( "STR.LST" );
ifs >> myStringList;

myStringList->get( legend, 1);

cout << "Legend 1: " << legend << endl;
// "Pass, Do not GO!" と表示します。
```

第10章
ヒントとコツ

この章では，Turbo Visionをより効果的に使うためのヒントをいくつか紹介します。オブジェクト指向プログラミングおよびイベント駆動型プログラミングは，熟練したプログラマにとっても比較的新しい概念なので，これについても多少説明しておきましょう。

Turbo Visionアプリケーションのデバッグ

このマニュアルで示したプログラム例をステップ動作させたり，トレースしてみたりしても，おそらくなかなか先に進むことができないということがおわかりになることと思います。Turbo Visionのプログラムはイベント駆動型であるため，プログラムの実行時間の多くは(場合によっては大半が)，何らかのイベントが発生するまで ***TGroup::execute*** 内のかなり小さなループで費やされることになります。したがって，そのループ内をステップ動作させたり，トレースさせたりしてみてもあまり役に立ちません。

☞ Turbo Visionのデバッグは，ひたすらブレークポイントをかけていくというのが秘訣です。

それでは，適当なところにブレークポイントをかけることによって，Turbo Visionプログラムのどこに問題があるかが発見しやすくなることを以下で見てみることにしましょう。

実行されないプログラム

アプリケーションをデバッグすると，プログラムのある部分が実

行されないという問題が出てきたりします。たとえばステータスラインのある項目をクリックしたり，メニューで選択を行なうことによってウィンドウが表示される**はず**なのに，そうならないといった場合です。

そこで普通は，そのコマンドのところまでプログラムをステップ実行させてみて，意図したところが実行されるかわりに，どこが**実行されている**のかを調べようと考えるわけですが，そうしたとしてもうまくいきません。ステップ実行させても，また元のところにもどってくるだけだからです。

このような場合，その実行されないプログラムを呼び出すことになっている ***handleEvent*** にブレークポイントを設定するのが一番うまいやり方です。***handleEvent*** の頭のところにブレークポイントを設定して，ブレークがかかったら，そのときのイベントオブジェクトを調べてみて，それが意図したイベントであるかどうかを確かめるわけです。ここまできたら，***handleEvent*** も，みなさんが定義したコマンドを処理するプログラムも，みなさんが記述したわけですから，ステップ動作させたり，トレースしたりすることができます。

イベントがマスクされている

ビューが，処理させたいイベントを受け取れないのには，いくつか理由が考えられます。まず，もっとも単純な間違いとしては，オブジェクトのイベントマスクでイベントの種類を正しく指定していないということが考えられます。そのオブジェクトに，ある特定のイベントを処理してもよいということを指示しておかないかぎり，オブジェクトはそのイベントを見ようともしません。

盗まれたイベント

2番目の可能性として，別のオブジェクトがそのイベントを“奪っていないかどうか”調べてみる必要があります。すなわち，そのイベントが意図したオブジェクト以外のオブジェクトによって処理され，クリアされているという場合です。

この場合にも，原因がいくつか考えられます。まず，コマンドを

二重定義している場合です。２つのコマンドに同じ定数値を割り当てた場合，その両方のコマンドが無差別的に処理されます。したがって，とくにコードモジュールを再利用する場合，どの定数にどういう値を割り当てたのかを確かめてみることが大切になってきます。

もう１つは，コマンドラベルを二重定義していることが考えられます。とくに，コードを再利用した場合です。たとえば，***cmJump*** というコマンドを定義するとき，コマンド ***cmJump*** を処理する ***handleEvent*** が別のオブジェクトにすでにあることを忘れて，それを削除せずにいたとしたら，衝突が起こることになります。イベントを“ちゃんと受けることができない”場合には，そのイベントを他のオブジェクトが処理していないかどうか確かめるようにしてください。

親のせい

最後に，イベントが，親オブジェクトによって処理されていないかどうかを確認する必要があります。導出した型の ***handleEvent*** メソッドが，イベントの大部分を親のイベントハンドラに任せることがよくあり，意図しなかったものまで処理させてしまう可能性もあります。その場合には，親の ***handleEvent*** を呼び出す前にそのイベントを獲得するようにしてください。

意図したとおりにならない場合

場合によっては，ウィンドウは表示されるのだけれども，ゴミが表示されたり，意図したものが表示されなかったりすることがあります。その場合，イベントは正しく処理されているわけですが，そのイベントに対応するプログラムが間違っているか，あるいはオーバーライドされていることになります。こういう場合には，イベントの発生によって呼び出される関数にブレークポイントを設定するのが一番良い方法です。ブレークがかかったら，普通にステップ動作させるなり，トレースすればよいでしょう。

ハングアップした場合

ハングアップを招くようなバグは，一番追いかけにくいバグですが，それでも発見できないというわけでは**ありません**。まず，上で示したやり方にしたがって，メンバ関数に何らかのブレークポイントをかけてみて，ハングアップの起こる場所を絞りこんでいきます。次に，ポインタが2回廃棄されていないかどうかを調べてみる必要があります。あるオブジェクトがオーナーから廃棄されたにもかかわらず，さらに直接そのオブジェクトを廃棄しようとした場合に，こういうことが起こる場合があります。

間違ったオブジェクト型にストリームデータを読み込んだり，コレクションのデータの型キャストを間違った場合などにも，ハングアップする可能性があります。

Turbo Vision への移植

既存のアプリケーションを Turbo Vision に移植したいという場合，まず Turbo Vision のインターフェースをアプリケーションに移植しようとか，アプリケーションの上に Turbo Vision のレイヤを被せようというようなことを考えてみるのではないでしょうか。しかしながら，これはフラストレーションを招くだけです。Turbo Vision のアプリケーションはイベント駆動型なので，既存のアプリケーションをこのプログラミングパラダイムに移行させることは，不可能ではないにしても，たいていの場合容易ではありません。

既存のプログラムの手直し

そこで，移植をもっと簡単に行なう方法があります。これまで，Turbo Vision で何らかのアプリケーションをプログラミングする場合，アプリケーションの ***draw*** および ***handleEvent*** が基本だということを学んできました。既存のプログラムを移植する場合のうまいやり方は，まず，その既存のプログラムに沿って，Turbo Vision のインターフェースを記述し，その新しいアプリケーション用に既存のプログラムを手直しするというやり方です。手直ししたプログラムは，たいてい，最終的には，新たに作成したビューの ***draw*** および ***handleEvent*** に適合するようになります。

アプリケーションのどの部分が本質的なのかを判断し，アプリケーションにおいて本質的な仕事を行なう部分とインターフェース部分とを分離するには，少し時間がかかります。アプリケーションを別の角度から捉える必要があるので，場合によっては困難な作業となることもあります。

移植作業には，新たに作成したオブジェクトに，自分自身をどう表現するかを教え込むための書き換え作業も含まれますが，場合によっては，既存のプログラムのインターフェース部分を破棄しなければならないことも多々あります。

アプリケーションの移植を行なってみると，プログラムの中の驚

くほど多くの部分がユーザインターフェース処理に費やされていたことに気付くことと思います。Turbo Visionにその作業を任せることによって，それまで行なってきたユーザインターフェース処理の大部分は，単に捨てればよいことになります。

われわれがボーランドの統合環境を Turbo Visionに移植した際，この方法が非常に効果的でした。コンパイラ，エディタ，デバッガなど，すべてのエンジンについて従来のユーザインターフェースを手直しし，Turbo Visionで記述されたユーザインターフェースに組み込みました。

仕組みを考え直して

この新しいパラダイムによるプログラミングに慣れるまでには，少し時間がかかります。従来のプログラミング法では，プログラムをコードの側から捉えがちでした。われわれがコードであり，データは“外側”にあって処理対象となる何物かである，という考え方です。直感的には，プログラム開発環境などのようなプログラムは，エディタオブジェクトを中心にして構成すればよいように思われます。統合環境では，ほとんどの時間は何といっても編集作業に費やされるわけですから。エディタで編集を行なったら，ときどきコンパイラを呼び出せばよいというわけです。

しかしながら，オブジェクト指向プログラミングの真価を発揮するためには，考え方をいろいろと変える必要があります。統合環境の場合，アプリケーション自体がオブジェクトを構成するようにしたほうが，理にかなったやり方だといえます。編集する必要があれば，アプリケーションはエディタを呼び出し，コンパイルする必要があればコンパイラを初期化し，どのファイルをコンパイルするのかをコンパイラに伝えます。

コンパイラがエラーを検出したとき，ユーザをソースコード内のエラー箇所に連れもどしたい場合はどうでしょうか。アプリケーションはコンパイラを呼び出し，コンパイラから結果を得ます。コンパイラが結果としてエラーを返してくる場合には，ファイル名と行番号もいっしょに返してきます。アプリケーションは，すでにそのファイルをエディタでオープンしているかどうか調べ，もしオープ

ンしていなかったら，オープンします。アプリケーションは，ライン番号などのエラー情報をエディタに渡し，エディタで使用するエラーメッセージ文字列を作成します。

エディタは，コンパイラについては何も関知する必要はなく，またコンパイラもエディタについては何ら関知する必要がありません。アプリケーション自体が全体の中心となります。エディタを必要とするのはアプリケーションであり，またコンパイラを必要とするのもアプリケーションです。結局，いろいろなものを結び付けるのは，アプリケーション以外の何物でもないというわけです。アプリケーションを単なるデータの塊だとみなし，どこか外に位置するものだとみなし続けるなら，アプリケーションの中心をどこか別のところに置いてしまったりすることになります。そうして，プログラム内の各部品間の過度の通信処理を無理やり行なう必要がでてくるわけです。

Turbo Vision で統合環境を記述するという作業には，何もかも含めて，一から作り始めるとすればかかるであろう時間の何分の一かの時間しかかかりませんでした。われわれは，みなさんが次にアプリケーションを作成するときには，この威力を実感できるものと期待しています。

ビットマップフィールドの使い方

Cに熟練したプログラマは，この節を読みとばしてもかまいません。

Turbo Visionは，**ビットマップ**のデータメンバをたくさん備えており，バイトもしくはワードの各ビットで，いろいろな特性を指定できるようになっています。それぞれのビットは，セット（=1）するかクリア（=0）するかで，指定した特性を有効にするか否かを決めることになりますので，一般に**フラグ**と呼ばれています。

たとえば，どのビューにも ***options*** と呼ばれるビットマップフィールドが備わっています。このワードの個々のビットは，それぞれTurbo Visionに対して別々の意味を持っています。***options*** ワードの各ビットの定義は第13章に解説されています。

フラグ値

各ビットにはニーモニックが割り当てられています。たとえば，4番目のビットは，***ofFramed*** と呼ばれるものです。***ofFramed*** が1にセットされていれば，そのビューでは周囲にフレームを表示することを意味します。このビットが0なら，そのビューにはフレームはつきません。

自分で独自のフラグを定義するつもりがなければ，フラグビットにどのような値が入っているかについてあれこれ考える必要はありません。また，独自のフラグを設ける場合でも，その定義が一義的なものであるようにすればよいだけです。たとえば，Turbo Visionの ***options*** ワードの上位6ビットは定義されていません。したがって導出したビューで，それらのビットをどのようにでも利用することができます。

ビットマスク

マスクは，複数のビットフラグをまとめて簡便に扱うためのものです。Turbo Visionでは，いろいろな種類のイベントに対してマス

クを定義しています。たとえば ***evMouse*** マスクには，マウスイベントのそれぞれの種類に対応した4つのビット全部が含まれます。したがって，ビューがマウスイベントを調べたい場合，イベントが ***evMouse*** マスクで指定されているものに含まれているかどうかを調べることで，イベントの種類を知ることができます。マウスイベントの種類それぞれについてチェックする必要はないということです。

ビット演算

CおよびC++では，フラグを操作したりテストするために，ビット演算子をいろいろと組み合わせて使用します。以下にいくつかの例を示します。

ビットのセット

ビットをセットするには，| 演算子を使用します。たとえば，*myButton* というボタンの **options** フィールドにある ***ofPostProcess*** ビットをセットするには，

```
myButton.options |= ofPostProcess;
// myButton.options = (myButton.options|ofPostProcess); と同じ
```

とします。

ビットを立てるために+を使用しないでください。たとえば，上のコードのかわりに，

こういうことは行なわないでください！

```
myButton.options += ofPostProcess;
```

というコードにしたとすると，***ofPostProcess*** ビットが0だった場合にかぎり，正しく動作します。このビットが立っていた場合，この2進加算を行なうと，次のビット（***ofBuffered***）にキャリーが生ずることになり，***ofBuffered*** をセットするか，クリアすることになります。セットされるか，クリアされるかは，***ofBuffered*** がクリアされていたか，セットされていたかによります。このようにして，上のフラグに波及していきます。

複数のビットを1つの文でセットすることもできます。以下のコードがその例であり，*myScroller* というスクロールビューの2つのグローモードフラグを一度にセットします。

```
myScroller.growMode |= (gfGrowHiX | gfGrowHiY);
```

ビットクリア

ビットのクリアには，**&** 演算子（ビットごとの AND）および ~ 演算子（ビットごとの否定）を使用します。*aLabel* というラベルの ***dragMode*** フィールドにある ***dmLimitLoX*** ビットをクリアするには，次のようにします。

```
aLabel.dragMode &= ~dmLimitLoX;
```

ビットセットと同様，複数のビットを1回の演算でクリアすることもできます。

ビット反転

^ 演算子（ビットごとの XOR）を利用すれば，ビットの反転を行なう（1を 0 に，0を1にする）ことができます。

```
myButton.options ^= ofPreProcess;
// ofPreProcess フラグを切り換える
myView ^= (growModeHiX | growModeHiY);
// 両方のフラグを切り換える
```

ビットテスト

ビューで，ある特定のフラグビットがセットされているかどうかを調べなければならなくなることがよく生じます。そのような処理には，**&** 演算が向いています。たとえば，デスクトップがウィンドウ *aWindow* をタイル表示してよいかどうかを調べる場合には，以下のようにして ***ofTileable*** オプションフラグを調べることになります。

```
if (aWindow.options & ofTileable) { ... }
```

マスクの使用

また & を使用して，複数個のマスクビットがセットされているかどうかを調べることもできます。たとえば，イベントオブジェクトにマウスイベントのどれかが入っているかどうかを調べたい場合には，以下のようにします。

```
if (event.what & evMouse) { ... }
```

まとめ

以下にビットマップ演算をまとめておきます。

ビットセット

```
field |= flag;
```

ビットクリア

```
field &= ~flag;
```

ビットの反転

```
field ^= flag;
```

フラグがセットされているかどうかのチェック

```
if (field & flag) { ... }
```

フラグがマスクのものであるかどうかのチェック

```
if (flag & mask) { ... }
```

第3部

Turbo Vision リファレンス

第11章
リファレンスの使い方

このTurbo Visionリファレンスでは，Turbo Vision階層標準のクラスおよびメンバ関数，Turbo Visionのアプリケーションを開発する際に使用することになるニーモニック名および種々の定数，構造体を説明しています。チュートリアルとはなっていません。

Turbo Visionのような複雑なクラスライブラリは，その性質上，いろいろな構成要素からなっています。そこで，同じことがらについての説明を何度も何度も繰り返すことを避けるために，アルファベット順の見出しをつけて，できるだけ完全な情報としてまとめました(第13章～16章)。その他のところで詳しく説明していない場合は，そこに出てくる各要素について詳細に説明してある箇所を参照するように指示してありますから，Turbo Visionの階層において実際どのように関係しているかによって，その内容がわかるようになっています。

必要な情報の見つけ方

第12章『**ヘッダファイルクロスリファレンス**』は，Turbo Visionを構成する各ヘッダファイルについての説明です。各ヘッダファイルで宣言されているすべての型，定数，変数および関数をリストにしてまとめました。

第13章『**クラスリファレンス**』は，Turbo Vision標準のすべてのクラス型についての解説であり，そのデータメンバおよびメンバ関数をすべて紹介しています。クラスはアルファベット順に配列してあり，クラス内のデータメンバおよびメンバ関数もアルファベット順に掲載してあります。

第14章『**エディタクラス**』と第15章『**標準ダイアログボックスの実現**』では，シンプルなエディタおよび標準的なダイアログボックス機能を用意するための拡張クラスについて解説します。

第16章『**グローバルリファレンス**』では，Turbo Visionで使われているすべてのグローバルな定数，変数，関数をアルファベット順にして解説しています。一般に，クラスもしくはクラスの一部でないものは，この第16章で見つけることができます。

第13章から第15章までの各クラスの解説では，各クラスに固有なことがらだけが説明されます。たいていのクラスは，データメンバやメンバ関数を他のクラスから継承しています。したがって，あるクラスのメンバ関数を知りたい場合には，まずそのクラスを調べ，そのクラスの説明にそのメンバ関数の見出しがなかったら，その親のクラスを調べるか索引を調べることになります。各クラスの説明では，一番最初にその基本クラスと導出クラスの関係図を示しています。

オブジェクトの概要

各オブジェクト（基本オブジェクト ***TObject*** と特殊なオブジェクトである ***TPoint*** および ***TRect*** 以外）は，親のオブジェクトのデータメンバとメンバ関数を継承します。新たに導出したオブジェクトの場合も，その基本クラスのメンバ関数とデータメンバを継承します。標準のオブジェクトの多くは，導出したオブジェクトでオーバーライドする**必要のある抽象**メンバ関数を備えています。それ以外のメンバ関数で，virtual のついているものは，通常はオーバーライドして使用することを意味します。その他，オーバーライドせずに，デフォルトとして使用することで，有用な機能を果たすメンバ関数もあります。

名前の付け方

Turbo Vision の標準のオブジェクト型には，すべて，名前にプレフィクスとしてニーモニックが付けられます。名前の最初の 1 文字で，オブジェクト型なのか，オブジェクトのストリーム登録構造体なのか，あるいはオブジェクトのカラーパレットなのかが識別できるようになっています。

- オブジェクト型は，**T** で始まります。〈例〉 ***TObject***
- ストリーム登録構造体は，**R** で始まります。〈例〉 ***RObject***
- カラーパレットは，**cp** で始まります。〈例〉 ***cpObject***
- メンバ関数およびデータメンバの場合，最初の何文字かは小文字で始め，それ以降は，メンバ名を構成する各ワードを大文字で始めます。〈例〉 ***handleEvent***，***hScrollBar***

Turbo Vision の定数には，すべて，プレフィクスとして使用方法を示す 2 文字のニーモニックが付けられます。

表 11.1
Turbo Vision で使用される定数

プレフィクス	意味	例
ap	アプリケーションのパレット	*apColor*
bf	ボタンフラグ	*bfNormal*
cm	コマンド	*cmQuit*
co	コレクションコード	*coOverFlow*
dm	ドラッグモード	*dmDragGrow*
ev	イベント定数	*evMouseDown*
gf	グローモードフラグ	*gfGrowLoX*
hc	ヘルプコンテキスト	*hcNoContext*
kb	キーボード定数	*kbAltX*
mb	マウスボタン	*mbLeftButton*
mf	メッセージボックス	*mfWarningch*
of	オプションフラグ	*ofTopSelect*
sb	スクロールバー	*sbLeftArrow*
sf	ステートフラグ	*sfVisible*
wf	ウィンドウフラグ	*wfMove*
wn	ウィンドウ番号	*wnNoNumber*
wp	ウィンドウパレット	*wpBlueWindow*

第12章
ヘッダファイル
クロスリファレンス

Turbo Visionでは，いろいろな標準クラスや例題用として作成された導出クラスをグループ化してサポートするために，ヘッダファイルが各種用意されています。例題用として作成された導出クラスの多くは，少し手直しすれば自分のアプリケーションに組み込むことができる専門化されたクラスとなっています。

この章は，Turbo Visionの各ヘッダファイルで，どのようなクラスが宣言されているかを素早く知るためのクロスリファレンスとなっています。また，クラス階層図によって，知りたいクラスが13～15章のうちのどの章で説明されているかわかるようになっています。各クラスの詳細については，13～15章を参照してください。13～15章では，全クラスをアルファベット順に並べてあり，また各クラス内でも，データメンバ，メンバ関数をアルファベット順に並べてあります。

ヘッダファイルには，それらしい名前を付けてありますが，あるクラスがどのヘッダファイルで定義されているかは必ずしも明白ではありません。**表12.1～12.3**は，各種ヘッダファイルを種類ごとにまとめたものです。これらの表には，各ヘッダファイルの内容がもう少し詳しく示してあります。

表 12.1
標準ヘッダファイル

ヘッダファイル	内容
APP	おもなアプリケーションクラスである ***TProgram, TApplication***
BUFFERS	ビデオメモリ処理
CONFIG	各種システム関連パラメータ
DIALOGS	ダイアログボックス，コントロール，ヒストリーの各クラス
DRAWBUF	表示バッファの各クラス
MENUS	メニュー，ステータスラインの各クラス
MSGBOX	メッセージ，入力ボックスの各グローバル
OBJECTS	ビューでない基本クラス（点，矩形，コレクション）
RESOURCE	リソースとリソースに関連する各クラス
SYSTEM	マウスおよびキーボードのイベント処理用の各クラス
TEXTVIEW	テキストデバイス用の特殊クラス
TKEYS	キーボード定数
TOBJSTRM	ストリームクラス
TTYPES	基本的な typedef と定数
TV	#include 制御用のマスタヘッダ
TVOBJS	***TObject*** およびストリーム入出力ができないコレクション
UTIL	各種グローバル
VIEWS	ビュー，グループ，ウィンドウ，フレーム，スクロールバーの各クラス

以下は，上に示した標準クラスに特殊な拡張を行なったものです。

表 12.2
特殊ヘッダファイル

ヘッダファイル	内容
COLORSEL	パレット選択クラス（定義は第 13 章にあります）
EDITORS	エディタ用に特化したクラス（定義は第 14 章にあります）
STDDLG	特化したダイアログボックスおよび入力行（定義は第 15 章にあります）

以下のクラスは，個々の目的に応じた処理を行なうときの標準クラスの活用法を示すために用意されたものです。

表 12.3
デモ用ヘッダファイル

ヘッダファイル	内容
ASCII	ASCII コード表のデモ
CALC	電卓のデモ
CALENDAR	カレンダのデモ
GADGETS	クロック表示およびヒープ表示のデモ
MOUSEDLG	マウスダイアロググクラス
FILEVIEW	ファイル表示クラス
PUZZLE	パズルのデモ
TVBGI	BGI のデモ

APP ヘッダファイル

APP ヘッダファイルでは，***TApplication***，***TBackground***，***TDeskInit***，***TDeskTop***，***TProgInit***，および ***TProgram*** の各クラスを定義しています。

BUFFERS ヘッダファイル

BUFFERSヘッダファイルでは，***TBufListEntry***，***TVideoBuf***，および ***TVMemMgr*** の各クラスを定義しています。また，グローバル定数 ***DEFAULT_SAFETY_POOL_SIZE*** の定義も行なっており，その初期値に 8,192を設定しています。

CONFIGヘッダファイル

CONFIGヘッダファイルは，内部的にしか使用されていません。以下の定数が定義されています。

表 12.4
CONFIG.H での定数定義

定数	定義
eventQSize	16
maxCollectionSize	(int)((65536uL – 16)/sizeof(void *))
maxFindStrLen	80
maxReplaceStrLen	80
maxViewWidth	132

DIALOGSヘッダファイル

DIALOGSヘッダファイルでは，***cmRecordHistory*** 定数，ボタン型の定数数個，およびダイアログボックスに関連する以下のクラスを定義しています。

TButton	*THistory*	*TListBox*
TCheckBoxes	*THistoryViewer*	*TParamText*
TCluster	*THistoryWindow*	*TRadioButtons*
TDialog	*TInputLine*	*TSItem*
THistInit	*TLabel*	*TStaticText*

表 12.5
DIALOGS.H での定数定義

定数	定義
bfNormal	0x00
bfDefault	0x01
bfLeftJust	0x02
bfBroadcast	0x04
cmRecordHistory	60

DRAWBUF ヘッダファイル

DRAWBUF ヘッダファイルでは，クラス ***TDrawBuffer***，およびロングワードから文字ワードや属性ワードを取り出すときに便利な ***loWord***，***hiWord*** マクロを定義しています。

MENUS ヘッダファイル

MENUS ヘッダファイルでは，***TSubMenu***，***TMenuItem***，***TStatusDef***，および ***TStatusItem*** でオーバーロードして使用する＋演算子を定義しています。また，以下のクラスの定義も行なっています。

TMenuBar	***TMenuItem***	***TStatusDef***	***TStatusLine***
TMenuBox	***TMenuView***	***TStatusItem***	***TSubMenu***

MSGBOX ヘッダファイル

MSGBOX ヘッダファイルでは，以下のものを定義しています。

表 12.6
MSGBOX.H で行なわれている各種定義

項目	値	意味
グローバル関数		
messageBox		
messageBoxRect		
inputBox		
inputBoxRect		
メッセージボックス定数		
mfWarning	0x0000	警告ボックスの表示
mfError	0x0001	エラーボックスの表示
mfInformation	0x0002	情報ボックスの表示
mfConfirmation	0x0003	確認ボックスの表示
メッセージボックスのボタンフラグ		**ダイアログボックスに入れるもの**
mfYesButton	0x0100	Yes ボタン
mfNoButton	0x0200	No ボタン
mfOKButton	0x0400	OK ボタン
mfCancelButton	0x0800	Cancel ボタン
mfYesNoCancel	*mfYesButton* | *mfNoButton* | *mfCancelButton*	標準的な Yes，No，Cancel ボタンを備えたダイアログボックス
mfOKCancel	*mfOKButton* | *mfCancelButton*	標準的な OK，Cancel ボタンを備えたダイアログボックス

OBJECTS ヘッダファイル

OBJECTSヘッダファイルでは，*TCollection, TPoint, TRect,* および *TSortedCollection* の各クラスを定義しています。

RESOURCE ヘッダファイル

RESOURCE ヘッダファイルでは，*TResourceCollection, TResourceFile*，*TStrIndexRec*，*TStringCollection*，*TStringList*，*TStrListMaker* の各クラス，および構造体 *TResourceItem* を定義しています。

SYSTEM ヘッダファイル

SYSTEM ヘッダファイルでは，以下の各クラスを定義しています。

Int11trap	***TMouse***
TDisplay	***TScreen***
TEventQueue	***TSystemError***
THWMouse	***Tkey***

また，構造体 ***CharScanType***，***MessageEvent***，***MouseEventType***，***TEvent*** の定義も行なっています。

さらに以下のイベントコードと外部変数の定義も行なっています。

表 12.7
SYSTEM.H で定義される変数とその値

項目	値	項目	値
イベントコード（定数）			
evMouseDown	0x0001；	***evKeyDown***	0x0010；
evMouseUp	0x0002；	***evCommand***	0x0100；
evMouseMove	0x0004；	***evBroadcast***	0x0200；
evMouseAuto	0x0008；		
イベントマスク（定数）			
evNothing	0x0000；	***evKeyboard***	0x0010；
evMouse	0x000f;	***evMessage***	0xFF00；
マウスボタン状態マスク（定数）			
mbLeftButton	0x01；		
mbRightButton	0x02；		

TEXTVIEW ヘッダファイル

TEXTVIEW ヘッダファイルでは，***TTextDevice*** と ***TTerminal*** のクラスを定義しています。

TKEYS ヘッダファイル

TKEYS ヘッダファイルでは，以下のものを定義しています。

表 12.8 キーボードの状態マスクとシフトマスク

マスク	値	対応するキー
kbRightShift	0x0001	***SHIFT*** （注）
kbLeftShift	0x0002	***SHIFT*** （注）
kbCtrlShift	0x0004	***CTRL***
KbAltShift	0x0008	***GRPH***
kbCapsState	0x0040	***CAPS***

（注）PC-9801 では，シフトキー状態を取得すると，どちらかのシフトキーが押されていれば，***kbRightShift*** と ***kbLeftShift*** の両方が ON になります。

表 12.9 キーコード

キーコード	値	対応するキー	キーコード	値	対応するキー
kbAlt0	0x8100	GRPH+0	*kbAltF4*	0x6b00	GRPH+F4
kbAlt1	0x7800	GRPH+1	*kbAltF5*	0x6c00	GRPH+F5
kbAlt2	0x7900	GRPH+2	*kbAltF6*	0x6d00	GRPH+F6
kbAlt3	0x7a00	GRPH+3	*kbAltF7*	0x6e00	GRPH+F7
kbAlt4	0x7b00	GRPH+4	*kbAltF8*	0x6f00	GRPH+F8
kbAlt5	0x7c00	GRPH+5	*kbAltF9*	0x7000	GRPH+F9
kbAlt6	0x7d00	GRPH+6	*kbAltF10*	0x7100	GRPH+F10
kbAlt7	0x7e00	GRPH+7	*kbAltEqual*	0x8300	GRPH+=
kbAlt8	0x7f00	GRPH+8	*kbAltMinus*	0x8200	GRPH+−
kbAlt9	0x8000	GRPH+9	*kbAltSpace*	0x0200	GRPH+SPACE
kbAltA	0x1e00	GRPH+A	*kbBack*	0x0e08	BS
kbAltB	0x3000	GRPH+B	*kbCtrlA*	0x0001	CTRL+A
kbAltC	0x2e00	GRPH+C	*kbCtrlB*	0x0002	CTRL+B
kbAltD	0x2000	GRPH+D	*kbCtrlC*	0x0003	CTRL+C
kbAltE	0x1200	GRPH+E	*kbCtrlD*	0x0004	CTRL+D
kbAltF	0x2100	GRPH+F	*kbCtrlE*	0x0005	CTRL+E
kbAltG	0x2200	GRPH+G	*kbCtrlF*	0x0006	CTRL+F
kbAltH	0x2300	GRPH+H	*kbCtrlG*	0x0007	CTRL+G
kbAltI	0x1700	GRPH+I	*kbCtrlH*	0x0008	CTRL+H
kbAltJ	0x2400	GRPH+J	*kbCtrlI*	0x0009	CTRL+I
kbAltK	0x2500	GRPH+K	*kbCtrlJ*	0x000a	CTRL+J
kbAltL	0x2600	GRPH+L	*kbCtrlK*	0x000b	CTRL+K
kbAltM	0x3200	GRPH+M	*kbCtrlL*	0x000c	CTRL+L
kbAltN	0x3100	GRPH+N	*kbCtrlM*	0x000d	CTRL+M
kbAltO	0x1800	GRPH+O	*kbCtrlN*	0x000e	CTRL+N
kbAltP	0x1900	GRPH+P	*kbCtrlO*	0x000f	CTRL+O
kbAltQ	0x1000	GRPH+Q	*kbCtrlP*	0x0010	CTRL+P
kbAltR	0x1300	GRPH+R	*kbCtrlQ*	0x0011	CTRL+Q
kbAltS	0x1f00	GRPH+S	*kbCtrlR*	0x0012	CTRL+R
kbAltT	0x1400	GRPH+T	*kbCtrlS*	0x0013	CTRL+S
kbAltU	0x1600	GRPH+U	*kbCtrlT*	0x0014	CTRL+T
kbAltV	0x2f00	GRPH+V	*kbCtrlU*	0x0015	CTRL+U
kbAltW	0x1100	GRPH+W	*kbCtrlV*	0x0016	CTRL+V
kbAltX	0x2d00	GRPH+X	*kbCtrlW*	0x0017	CTRL+W
kbAltY	0x1500	GRPH+Y	*kbCtrlX*	0x0018	CTRL+X
kbAltZ	0x2c00	GRPH+Z	*kbCtrlY*	0x0019	CTRL+Y
kbAltF1	0x6800	GRPH+F1	*kbCtrlZ*	0x001a	CTRL+Z
kbAltF2	0x6900	GRPH+F2	*kbCtrlBack*	0x0e7f	CTRL+BS
kbAltF3	0x6a00	GRPH+F3	*kbCtrlDel*	0x0600	CTRL+DEL

kbCtrlEnd	0x7500	***CTRL+HELP***	***kbF7***	0x4100	***F7***
kbCtrlEnter	0x1c0a	***CTRL+*** ⏎	***kbF8***	0x4200	***F8***
kbCtrlHome	0x7700	***CTRL+HOME/CLR*** (注)	***kbF9***	0x4300	***F9***
kbCtrlIns	0x0400	***CTRL+INS***	***kbF10***	0x4400	***F10***
kbCtrlPgDn	0x7600	***CTRL+ROLLUP***	***kbGrayMinus***	0x4a2d	−(テンキー)
kbCtrlPgUp	0x8400	***CTRL+ROLLDOWN***	***kbGrayPlus***	0x4e2b	+(テンキー)
kbCtrlLeft	0x7300	***CTRL+***←	***kbHome***	0x4700	***HOME/CLR*** (注)
kbCtrlRight	0x7400	***CTRL+***→	***kbIns***	0x5200	***INS***
kbCtrlF1	0x5e00	***CTRL+F1***	***kbLeft***	0x4b00	←
kbCtrlF2	0x5f00	***CTRL+F2***	***kbNoKey***	0x0000	
kbCtrlF3	0x6000	***CTRL+F3***	***kbPgDn***	0x5100	***ROLLUP***
kbCtrlF4	0x6100	***CTRL+F4***	***kbPgUp***	0x4900	***ROLLDOWN***
kbCtrlF5	0x6200	***CTRL+F5***	***kbRight***	0x4d00	→
kbCtrlF6	0x6300	***CTRL+F6***	***kbShiftDel***	0x0700	***SHIFT+DEL***
kbCtrlF7	0x6400	***CTRL+F7***	***kbShiftF1***	0x5400	***SHIFT+F1***
kbCtrlF8	0x6500	***CTRL+F8***	***kbShiftF2***	0x5500	***SHIFT+F2***
kbCtrlF9	0x6600	***CTRL+F9***	***kbShiftF3***	0x5600	***SHIFT+F3***
kbCtrlF10	0x6700	***CTRL+F10***	***kbShiftF4***	0x5700	***SHIFT+F4***
kbDel	0x5300	***DEL***	***kbShiftF5***	0x5800	***SHIFT+F5***
kbDown	0x5000	↓	***kbShiftF6***	0x5900	***SHIFT+F6***
kbEnd	0x4f00	***HELP***	***kbShiftF7***	0x5a00	***SHIFT+F7***
kbEnter	0x1c0d	⏎	***kbShiftF8***	0x5b00	***SHIFT+F8***
kbEsc	0x011b	***ESC***	***kbShiftF9***	0x5c00	***SHIFT+F9***
kbF1	0x3b00	***F1***	***kbShiftF10***	0x5d00	***SHIFT+F10***
kbF2	0x3c00	***F2***	***kbShiftIns***	0x0500	***SHIFT+INS***
kbF3	0x3d00	***F3***	***kbShiftTab***	0x0f00	***SHIFT+TAB***
kbF4	0x3e00	***F4***	***kbTab***	0x0f09	***TAB***
kbF5	0x3f00	***F5***	***kbUp***	0x4800	↑
kbF6	0x4000	***F6***			

(注) XA, XL2 では，***CLR*** または ***HOME*** キー。それ以外は ***HOME/CLR***。

TOBJSTRM ヘッダファイル

TOBJSTRMヘッダファイルでは，以下の各クラスを定義しています。

fpbase	***pstream***
fpstream	***TPReadObjects***
ifpstream	***TPWObj***
iopstream	***TPWrittenObjects***
ipstream	***TStreamable***
ofpstream	***TStreamableClass***
opstream	***TStreamableTypes***

またこのヘッダファイルでは ***BUILDER*** およびマクロ ***__link***, ***__DELTA*** の定義も行なっています。

TTYPES ヘッダファイル

TTYPESヘッダファイルでは，以下の定義および設定を行なっています。

- const ccNotFound = -1 ;
- const char EOS = '¥0';
- enum Boolean { False, True } ;
- enum StreamableInit { streamableInit } ;
- extern const uchar specialChars [] ;
- typedef Boolean (* ccTestFunc)(void *, void *) ;
- typedef int ccIndex;
- typedef unsigned char uchar;
- typedef unsigned short ushort;
- typedef unsigned long ulong ;
- typedef void (* ccAppFunc)(void *, void *) ;

TV ヘッダファイル

TV.H は,「インクルード制御用」のヘッダファイルとして使用されます。

TV ヘッダファイルは，アプリケーションプログラムで必要なすべてのヘッダファイルを，簡便にかつ重複することのないように #include するためのものです。TV.H は，CONFIG.H や UTIL.H などの Turbo Vision で絶対必要なヘッダを必ず #include します。その他のヘッダファイルは, #if defined 指令を使用して選択的にインクルードされます。

TV.H は，識別子 *Uses_ClassName* (*ClassName* はクラス名かクラス登録名）が定義済みであるかどうかを検査します。検査の結果に従って，クラス *ClassName* のヘッダファイルおよびその基本クラスを自動的にインクルードします。プログラムでクラス *ClassName* を使用したい場合には，そのプログラムまたはヘッダに #define Uses_ClassName の 1 行を入れ，その下に #include <tv.h>の 1行を入れるだけで済むようになります。これによって，そのクラスおよび基本クラスの取り込み処理を自動化することができます。たとえば，プログラムの先頭で以下のように記述されていたとします。

```
#define Uses_TApplication
#include <tv.h>
```

そうすると，TV.H は以下のような検査を行ないます。

```
#if defined( Uses_TApplication )
#define Uses_TProgram
#define __INC_APP_H
#endif
```

TProgram は，***TApplication*** の基本クラスです。TV.H は，後で __INC_APP_H を，次のように検査します。

```
#if defined( __INC_APP_H )
#include <App.h>
```

最終的には，***TProgram***, ***TProgInit***, および ***TApplication*** のクラスの宣言を行なっている APP.H が，プログラムにインクルー

ドされることになります。以下のように基本クラスと導出クラスの両方を #define したとしても何の問題も起きません。

```
#define Uses_TProgram
#define Uses_TApplication
#include <tv.h>
```

ストリーム可能なクラスのストリーム登録は，_ _*link* マクロを使用して行なうことができます。たとえば，***TEditWindow*** を登録したい場合には，プログラムに以下の 1行を入れる必要があります。

```
#include <tv.h>
_ _link(REditWindow);
```

_ _***link*** マクロは，次のような extern 宣言を作り出します。

```
extern TStreamableClass REditWindow;
```

これによって，ストリームの読み書きを行なう関数が適切にリンクされることになります (第 13 章にある ***TStreamableClass*** の説明も参照してください)。

重要！ 独自のクラスを作成してファイルのインクルードを行なう場合，TV.H のやり方を基にして独自のインクルード制御ヘッダを作成することも可能です。ただし，TV.H の手直しは決してお薦めできません。というのは，TV.H は Turbo Vision の標準的な階層をカプセル化しているからです。したがって，TV.H に対して変更を行なうと，少しおかしくなったり，あるいはまったく悲惨な結果を招く可能性があります。

TVOBJS ヘッダファイル

TVOBJS ヘッダファイルでは，中核となる 3 個のクラス，すなわち ***TObject***, ***TNSCollection***, および ***TNSSortedCollection*** が定義されています。

VIEWS ヘッダファイル

VIEWS ヘッダファイルでは，以下の各クラスと，構造体が 1 つ定義されています。

TCommandSet	***TScroller***
TFrame	***TView***
TGroup	***TWindow***
TListViewer	***TWindowInit***
TPalette	***write_args***（構造体）
TScrollBar	

また，以下の標準コマンドコードを，以下に示した値で定義しています。

表 12.10 VIEWS.H で定義されているコードの値

コード	値	コード	値
cmValid	0	***cmZoom***	5
cmQuit	1	***cmResize***	6
cmError	2	***cmNext***	7
cmMenu	3	***cmPrev***	8
cmClose	4	***cmHelp***	9

また，以下の各種マスク，コマンド，コード，およびオプションも定義されています。

表 12.11　VIEWS.H の各種定数の値

変数	値	変数	値
TDialog 標準コマンド			
cmOK	10	*cmNo*	13
cmCancel	11	*cmDefault*	14
cmYes	12		
TView state マスク：			
sfVisible	0x001	*sfFocused*	0x040
sfCursorVis	0x002	*sfDragging*	0x080
sfCursorIns	0x004	*sfDisabled*	0x100
sfShadow	0x008	*sfModal*	0x200
sfActive	0x010	*sfDefault*	0x400
sfSelected	0x020	*sfExposed*	0x800
TView options マスク：			
ofSelectable	0x001	*ofBuffered*	0x040
ofTopSelect	0x002	*ofTileable*	0x080
ofFirstClick	0x004	*ofCenterX*	0x100
ofFramed	0x008	*ofCenterY*	0x200
ofPreProcess	0x010	*ofCentered*	0x300
ofPostProcess	0x020		
TView growMode マスク			
gfGrowLoX	0x01	*gfGrowHiY*	0x08
gfGrowLoY	0x02	*gfGrowAll*	0x0f
gfGrowHiX	0x04	*gfGrowRel*	0x10
TView dragMode マスク：			
dmDragMove	0x01	*dmLimitHiX*	0x40
dmDragGrow	0x02	*dmLimitHiY*	0x80
dmLimitLoX	0x10	*dmLimitAll*	*dmLimitLoX* | *dmLimitLoY* | *dmLimitHiX* | *dmLimitHiY*
dmLimitLoY	0x20		

TView ヘルプコンテキスト・コード：

hcNoContext	0		
hcDragging	1		

TScrollBar 部品コード：

sbLeftArrow	0	*sbDownArrow*	5
sbRightArrow	1	*sbPageUp*	6
sbPageLeft	2	*sbPageDown*	7
sbPageRight	3	*sbIndicator*	8
sbUpArrow	4		

TWindow::StandardScrollBar 用の *TScrollBar* オプション

sbHorizontal	0x000
sbVertical	0x001
sbHandleKeyboard	0x002

TWindow フラグマスク：

wfMove	0x01
wfGrow	0x02
wfClose	0x04
wfZoom	0x08

TWindow 番号定数：

wnNoNumber	0

TWindow パレット エントリ：

wpBlueWindow	0
wpCyanWindow	1
wpGrayWindow	2

Application コマンドコード：

cmCut	20	*cmClear*	24
cmCopy	21	*cmTile*	25
cmPaste	22	*cmCascade*	26
cmUndo	23		

表 12.11　VIEWS.H の各種定数の値（続き）

標準メッセージ：	
cmReceivedFocus	50
cmReleasedFocus	51
cmCommandSetChanged	52
TScrollBar メッセージ：	
cmScrollBarChanged	53
cmScrollBarClicked	54
TWindow 選択メッセージ：	
cmSelectWindowNum	55
TListViewer メッセージ：	
cmListItemSelected	56
イベントマスク：	
positionalEvents	*evMouse*
focusedEvents	***evKeyboard*** | ***evCommand***
typedef：	
typedef uchar *TScrollChars* [5]；	

クラス階層図

この節では，これまでの章に出てこなかったクラス階層図を紹介します。クラス階層の概要を表わす図は，99ページ，100ページに示してあります。これらの図の中で他と何も接続のないクラスは，その階層と近い関係にはあるがその階層の一部ではないクラスです。それらのクラスの近くにあるクラスと近い関係にあるクラスです。C＋＋の慣例に従って，矢印は基本クラスに**向かう**ことを表わします。

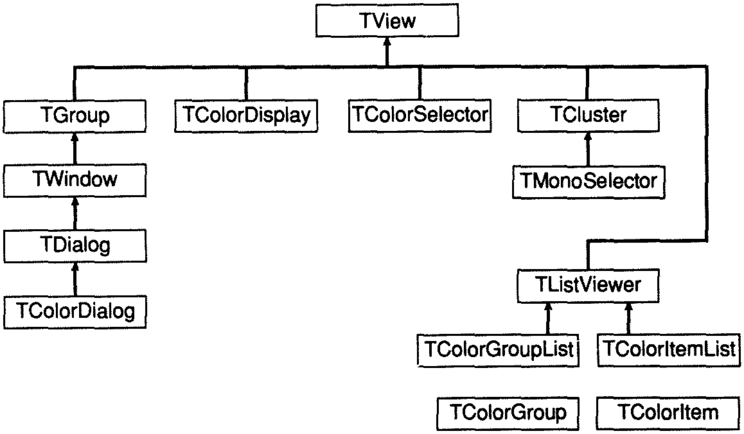


図 12.1
ビュアーとダイアログボックス

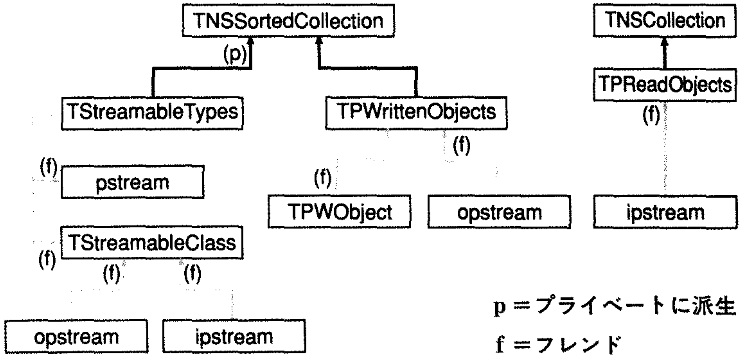
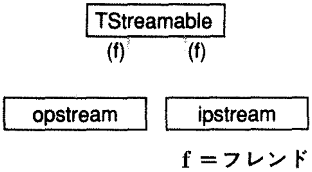


図 12.2
ストリーム入出力可能なクラス

図 12.3
TStreamable の
フレンド



第 13 章
クラスリファレンス

この章では，Turbo Vision のすべての標準クラスをアルファベット順に並べ，一般的な用途とその使用法，データメンバ，メンバ関数，およびカラーパレットについて解説します。また Turbo Vision で定義されている有用な構造体についても説明しています。第 12 章には，この章および 14 章，15 章に出てくるクラスのクロスリファレンスがあります。

この章の見方

クラスの情報を調べる場合には，階層内にあるクラスの性質は，その多くが基本クラスから継承されていることに注意してください。この章では，個々のクラスに関するすべての情報を際限なく記述するのではなく，**新規の**データメンバおよびメンバ関数，あるいはそのクラスで**変更された**データメンバおよびメンバ関数についてのみ記述しています。

検索しやすいように，すべてのメンバを索引に入れています。

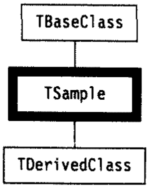
たとえば ***TLabel*** クラスのデータメンバ ***owner*** に関して調べたい場合，索引から ***owner*** を引くか，***TLabel*** のデータメンバのところを探してください。***owner*** は継承されたものなので，***TLabel*** のところには ***owner*** はありません。そこで，この階層で ***TLabel*** の直接の親（基本クラス）となっている ***TStaticText*** を調べてみることになりますが，そこにも ***owner*** はありません。さらに，***TStaticText*** の直接の親である ***TView*** を調べます。ここで初めて，変更されずに ***TLabel*** のところまで継承されてきた ***owner*** について情報を得ることができます。

この章では，各クラスの説明において，そのクラスの基本クラス

と直接の導出クラスを図示しています。したがって，どのクラスからメンバを継承しているかが簡単にわかるようになっています。

各クラスの説明は，以下の形式をとっています。

TSample クラス　　header.h



ここには，そのクラスについての簡単な概要が示されます。場合によっては，この情報だけで用が足りることもあるでしょう。

データメンバ

ここでは，それぞれのクラスの全データメンバをアルファベット順にして示します。データメンバの宣言および用途についての説明が示されます。

➡ データメンバおよびメンバ関数は，public，private，protected のうちのいずれかになります。protected である場合には，コードのところでそのことを示します。何も示されていない場合には，そのメンバは public であると考えてください（private のメンバについては，このマニュアルには何も記述していません)。

aDatamember　SomeType aDatamember ;

aDatamember は，このサンプルクラスのデータを格納するデータメンバです。その機能，意味，使い方について，この部分で説明します。

関連項目　関連のある**データメンバ**，**メンバ関数**，**クラス**，**グローバル関数**，等。

anotherDatamember ushort anotherDatamember ; `protected`

anotherDatamember についても，***aDatamember*** と同様の説明が行なわれます。

メンバ関数

ここには，このクラスで新たに定義したメンバ関数，または継承したメンバ関数をオーバーライドしたメンバ関数を示します。仮想メンバ関数については，それが，**常に**オーバーライドする必要のあるメンバ関数である場合には，そのことを明示的に示します。そうでない場合には，必要に応じてオーバーライドしてかまいません。

コンストラクタ ClassName (SomeType aParameter) ;

コンストラクタ ***ClassName*** は，クラス ***ClassName*** のオブジェクトを生成し，データメンバ ***aDatamember*** に *aParameter* を設定します。

zilch virtual void zilch ( ) ;

zilch は，サンプルクラスに何らかの処理を行なわせます。

関連項目 ***TSomethingElse::zilch***

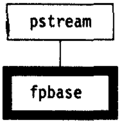
CharScanType SYSTEM.H

構造体 ***CharScanType*** は，キーストロークイベントの内容を表わすデータ（キャラクタコードとスキャンコード）を格納します。

```
struct CharScanType
{
   uchar charCode;
   uchar scanCode;
};
```

関連項目 ***KeyDownEvent*, *TEvent***

fpbase　　TOBJSTRM.H



fpbase は，すべてのオブジェクトのストリーム入出力に共通する基本的な操作を行ないます。

メンバ関数

コンストラクタ　fpbase ( ) ;

fpbase (const char *name, int omode, int prot = filebuf::openprot) ;

fpbase (int f) ;

fpbase (int f, char *b, int len) ;

バッファを使用する ***fpbase*** オブジェクトを新たに作成します。バッファのサイズと初期値は，引数 *len* と *b* で設定します。引数に *name*，*mode*，および保護 (*prot*) を指定するか，ファイルディスクリプタ *f* を指定することによってファイルをオープンし，そのファイルをストリームにアタッチすることができます。

デストラクタ　~fpbase ( ) ;

fpbase オブジェクトを消去します。

attach　void attach (int f) ;

ディスクリプタ *f* のファイルを，このストリームにアタッチすることができる場合にアタッチします。それに伴い ***ios::state*** を設定します。

close　void close ( ) ;

ストリームとそれにアタッチされているファイルをクローズします。

open　void open (const char *name, int mode,
int prot = filebuf::openprot) ;

指定された名前のファイルを，指定された *mode* (***app***, ***ate***, ***in***, ***out***, ***binary***, ***trunc***, ***nocreate***, ***noreplace***)および保護でオープンします。オープンされたファイルは，このストリームにアタッチされます。

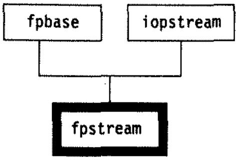
関連項目　第 8 章『**ストリーム入出力可能なオブジェクト**』

rdbuf　filebuf *rdbuf ( ) ;

現在のファイルバッファへのポインタを返します。

setbuf　void setbuf (char *buf, int len) ;

サイズが *len* のバッファを割り当てます。



fpstream は，基本である ***fpbase*** と ***iopstream*** を単に「ミックス」したものです。双方向ファイルストリームとの間で，ストリーム入出力可能なオブジェクトを同時に書き込んだり，読み出したりするための基本クラスです。Borland C++ ストリームライブラリの fstream.h で定義されている ***fstream*** と同様な働きをします。

メンバ関数

コンストラクタ

fpstream ( ) ;
fpstream (const char name *, int mode,
　　　　int prot = filebuf::openprot);
fpstream (int f);
fpstream (int f, char * b, int len) ;

バッファを使用する ***fpbase*** オブジェクトを新たに作成します。バッファのサイズと初期値は，引数 *len* と *b* で設定します。引数に name，mode，および保護を指定するか，ファイルディスクリプタ *f* を指定することによってファイルをオープンし，そのファイルをストリームにアタッチすることができます。

デストラクタ

~fpstream ( ) ;

fpstream オブジェクトを消去します。

open void open (const char * name, int mode,
int = prot filebuf::openprot) ;

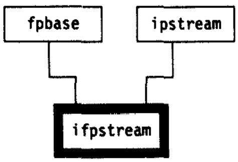
指定された名前のファイルを，指定された *mode* (***app***, ***ate***, ***in***, ***out***, ***binary***, ***trunc***, ***nocreate***, ***noreplace***)および保護でオープンします。オープンされたファイルは，このストリームにアタッチされます。

関連項目　第 8 章『**ストリーム入出力可能なオブジェクト**』

rdbuf filebuf *rdbuf ( ) ;

データメンバ ***bp*** を返します。

ifpstream　　TOBJSTRM.H



ifpstream は，基本である ***fpbase*** と ***ipstream*** を単に「ミックス」したものです。ファイルストリームから，ストリーム入出力可能なオブジェクトを読み出す（取り出す）ための基本クラスです。

メンバ関数

コンストラクタ

ifpstream ( ) ;
ifpstream (const char *name, int mode = ios::in,
int prot = filebuf::openprot) ;
ifpstream (int f) ;
ifpstream (int f, char *b, int len) ;

バッファを使用する ***ifpstream*** オブジェクトを新たに作成します。バッファのサイズと初期値は，引数 *len* と *b* で設定します。引数に

name, mode, および保護を指定するか，ファイルディスクリプタ *f* を指定することによって，ファイルをオープンし，そのファイルをストリームにアタッチすることができます。

デストラクタ ~ifpstream ( ) ;

Ifpstream オブジェクトを消去します。

open void open (const char *name, int mode = ios::in,
int prot = filebuf::openprot) ;

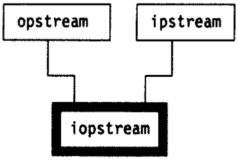
指定された名前のファイルを，指定された *mode* (***app***, ***ate***, ***in***, ***out***, ***binary***, ***trunc***, ***nocreate***, ***noreplace***) および保護でオープンします。*mode* のデフォルトは，保護が *openprot* の *in*(入力)です。オープンされたファイルは，このストリームにアタッチされます。

関連項目　第 8 章『**ストリーム入出力可能なオブジェクト**』

rdbuf filebuf *rdbuf ( ) ;

現在のファイルバッファへのポインタを返します。

iopstream TOBJSTRM.H



iopstream は，基本である ***opstream*** と ***ipstream*** を単に「ミックス」したものです。ストリーム入出力可能なオブジェクトを同時に読み書きするための基本クラスです。

メンバ関数

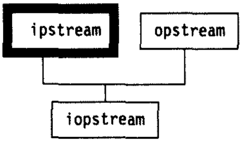
コンストラクタ　iopstream (streambuf *buf) ;

指定したバッファを使用する ***iopstream*** を新たに作成し，データメンバ ***bp*** に ***buf*** を設定します。状態は 0 に設定されます。

デストラクタ　~iopstream ( ) ;

iopstream オブジェクトを消去します。

ipstream　TOBJSTRM.H



ipstream は，***pstream*** から派生した入力専門のストリームであり，ストリーム入出力可能なオブジェクトを読み出す（取り出す）ための基本クラスです。Borland C++ ストリームライブラリの iostream.h で定義されている ***istream*** と同様の働きをします。***ipstream*** は，***TPReadObjects*** のフレンドクラスです。

メンバ関数

コンストラクタ　ipstream (streambuf *) ;

指定したバッファを使用する ***ipstream*** を新たに作成し，データメンバ ***bp*** に ***buf*** を設定します。状態は 0 に設定されます。

デストラクタ　virtual ~ipstream ( ) ;

ipstream オブジェクトを消去します。

find　const void *find (Pidtype id) ;　protected

id に対応するオブジェクトへのポインタを返します。

readByte　uchar readByte ( ) ;

ストリームの現在位置にある文字を返します。

readBytes　void readBytes (void * data, sizet sz) ;

ストリームの現在位置から *sz* バイト分読み出し，*data* に書き出します。

readData　void *readData (const TStreamableClass *c, void *mem) ;
protected

適当な ***read*** 関数を呼び出し，ストリームからオブジェクト **mem* への読み出しを行ないます。*mem* が 0 の場合には，まず適当な ***build*** 関数が呼び出されます。

関連項目　***TStreamableClass***，およびストリーム入出力可能なクラスそれぞれにあるメンバ関数 ***read***，***build***

readPrefix　const TStreamableClass *readPrefix ( ) ;　protected

現在位置に格納されているクラス ***name*** に対応する ***TStreamableClass*** オブジェクトを返します。

readString　char *readString ( ) ;
char *readString (char *buf, unsigned maxLen) ;

ストリームの現在位置から読み出された文字列を返します。

readSuffix　void readSuffix ( ) ;　protected

オブジェクトの名前フィールドの最終バイトを読み出し，検査を行ないます。

関連項目　***ipstream::readPrefix***

readWord　ushort readWord ( ) ;

ストリームの現在位置にあるワードを返します。

registerObject void registerObject (const void *adr) ; protected

オブジェクト **adr* のクラスを登録します。

seekg ipstream& seekg (streampos pos) ;
ipstream& seekg (streamoff off, seekdir dir) ;

1番目の形式は,ストリームの位置を *pos* で指定された絶対位置に移動します。2番目の形式は, *dir* で指定される現在位置から相対的にオフセット *off* (+または−) だけ離れた位置に移動します。*dir* には, ***beg*** (ストリームの先頭), ***cur*** (ストリームの現在位置), または ***end*** (ストリームの最後) を指定することができます。

tellg streampos tellg ( ) ;

ストリームの (絶対的な) 現在位置を返します。

I Std class

フレンド

operator >>

```
friend ipstream& operator >> ( ipstream& ps, signed char& ch );
friend ipstream& operator >> ( ipstream& ps, unsigned char& ch );
friend ipstream& operator >> ( ipstream& ps, signed short& sh);
friend ipstream& operator >> ( ipstream& ps, unsigned short& sh);
friend ipstream& operator >> ( ipstream& ps, signed int& i);
friend ipstream& operator >> ( ipstream& ps, unsigned int& i);
friend ipstream& operator >> ( ipstream& ps, signed long& l);
friend ipstream& operator >> ( ipstream& ps, unsigned long& l);
friend ipstream& operator >> ( ipstream& ps, float& f);
friend ipstream& operator >> ( ipstream& ps, double&d);
friend ipstream& operator >> ( ipstream& ps, long double& d);
friend ipstream& operator >> ( ipstream& ps, TStreamable& t);
friend ipstream& operator >> ( ipstream& ps, void * & t);
```

ipstream である *ps* から取り出し(読み出し)を行ない, 指定され

た引数に格納します。通常どおり >> 演算をチェインできるように，ストリームへの参照が返されます。引数のデータ型にしたがって，読み出す方法が決まります。たとえば signed char の読み出しには ***readByte*** が使用されます。

KeyDownEvent　　　　SYSTEM.H

KeyDownEvent

構造体 ***KeyDownEvent*** は，***keyCode***（***ushort*** 型）および ***charScan***（***struct CharScanType*** 型）の共用体です。この2つのメンバは，同じデータに対する2通りの捉え方，すなわちスキャンコードまたは unsigned を表わしています。スキャンコードは，プログラムがキーボードから受け取るときに使用され，unsigned は，switch 文で処理するときに使用されます。

```
struct KeyDownEvent
{
   union
   {
      ushort keyCode;
      CharScanType charScan;
   };
};
```

関連項目　***TEvent***，***TEvent::getKeyEvent***

MessageEvent　　　　SYSTEM.H

MessageEvent

構造体 ***MessageEvent*** の定義は以下のとおりです。

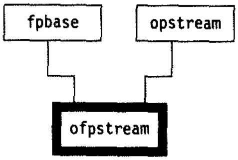
```
struct MessageEvent
{
   ushort command;
   union
```

```
    {
        void *infoPtr;
        long infoLong;
        ushort infoWord;
        short infoInt;
        uchar infoByte;
        char infoChar;
    };
};
```

メッセージイベントは，*command* によって内容の指定されるコマンドであり，その他に，1 バイトのデータから汎用ポインタまでのうちのいずれかの種類の情報が付加されます。このような構成とすることによって，Turbo Vision のオブジェクトが別の Turbo Vision のオブジェクトとの間でメッセージを送信したり受信したりする場合に，いろいろな用途のメッセージとすることができます。

関連項目　***TEvent***

ofpstream　TOBJSTRM.H



ofpstream は，基本である ***fpbase*** と ***opstream*** を単に「ミックス」したものです。ストリーム入出力可能なオブジェクトをファイルストリームに書き出す（挿入する）ための基本クラスです。

メンバ関数

コンストラクタ

```
ofpstream ( ) ;
ofpstream ( const char *name, int mode = ios::out,
        int prot = filebuf::openprot ) ;
ofpstream ( int f ) ;
```

O
Std class

ofpstream (int f, char *b, int len) ;

バッファを使用する ***ofpstream*** オブジェクトを新たに作成します。バッファのサイズと初期値は，引数 *len* と *b* を使用して設定します。引数に *name*, *mode*, および保護 (*prot*) を指定するか，ファイルディスクリプタ *f* を指定することによってファイルをオープンし，そのファイルをストリームにアタッチすることができます。

デストラクタ ~ofpstream ( ) ;

ofpstream オブジェクトを消去します。

open void open (const char * name,int mode = ios::out,
int prot = filebuf::openprot) ;

指定された名前のファイルを，指定された *mode* (***app***, ***ate***, ***in***, ***out***, ***binary***, ***trunc***, ***nocreate***, ***noreplace***) および保護でオープンします。モードのデフォルトは，保護が *openprot* の *out*(出力)です。オープンされたファイルは，このストリームにアタッチされます。

関連項目　第 8 章『**ストリーム入出力可能なオブジェクト**』

rdbuf filebuf * rdbuf ( ) ;

現在のファイルバッファを返します。

Operators

ストリーム入出力可能なクラスとその階層については，第 8 章の 242 ページおよび第 12 章の 307 ページを参照してください。

ストリーム入出力可能なすべてのクラスで，4 種類の演算子が宣言されています。すなわち，>> 演算子と << 演算子のそれぞれについて 2 種類ずつです。これらの演算子は，必要でない場合もよくありますが，多重継承が行なわれる場合でも，あいまいさが残らないようになっています。

>> 演算子が 2 個あった場合，その機能は，最初の演算子が ***TClassName*** オブジェクトへの**参照**をとり，2 番目の演算子が ***TClassName*** オブジェクトへの**参照へのポインタ**をとるという点で異なっていることに注意してください。同様に，<< 演算子が 2 個

あった場合，最初の演算子が ***TClassName*** オブジェクトへの**参照**をとり，2 番目の演算子が ***TClassName*** オブジェクトへの**ポインタ**をとるという点で，2つの演算子の機能は異なります。

operator >> ipstream& operator >> (ipstream& is , TClassName& cl) ;
ipstream& operator >> (ipstream& is , TClassName * & cl) ;

入力ストリーム *is* から ***TClassName*** オブジェクトを読み出し，それを *cl* に書き込みます。通常どおりに, >> 演算子のチェインが行なえるように，ストリームへの参照が返されます。

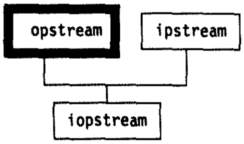
関連項目 ***ipstream***

operator << opstream& operator << (opstream& os , TClassName& cl) ;
opstream& operator << (opstream& os , TClassName *cl) ;

TClassName オブジェクトである *cl* を出力ストリーム *os* に書き込みます。通常どおりに，<< 演算子のチェインが行なえるように，ストリームへの参照が返されます。

関連項目 ***opstream***

opstream TOBJSTRM.H



opstream は，***pstream*** から派生した特殊化されたクラスであり，ストリーム入出力可能なオブジェクトを書き出す（挿入する）ための基本クラスです。***opstream*** は ***TPWrittenObjects*** のフレンドクラスです。

メンバ関数

コンストラクタ

```
opstream ( streambuf  * buf ) ;
opstream ( ) ;                                   protected
```

最初の形式は，指定されたバッファを使用する ***opstream*** を新たに作成し，データメンバ ***bp*** に *buf* を設定します。状態は 0 に設定されます。2 番目の形式は，デフォルトのバッファを割り当てます。

デストラクタ

```
~opstream ( ) ;
```

opstream オブジェクトを消去します。

find

```
P_id_type find ( const void  *adr ) ;            protected
```

オブジェクト **adr* のタイプ ID を返します。

flush

```
ostream& flush ( ) ;
```

ストリームをフラッシュします。

registerObject

```
void registerObject ( const void  *adr ) ;       protected
```

オブジェクト **adr* のクラスを登録します。

seekp

```
opstream& seekp ( streampos pos ) ;
opstream& seekp ( streamoff off, seekdir dir ) ;
```

1 番目の形式は，ストリームの現在位置を *pos* で指定された絶対位置に移動します。2 番目の形式は，*dir* で指定される現在位置から相対的にオフセット *off* （＋または－）だけ離れた位置に移動します。*dir* には，***beg***（ストリームの先頭），***cur***（ストリームの現在位置），または ***end*** （ストリームの最後）を指定することができます。

tellp

```
streampos tellp ( ) ;
```

ストリームの（絶対的な）現在位置を返します。

writeByte void writeByte (uchar ch) ;

ストリームにバイト *ch* を書き込みます。

writeBytes void writeBytes (const void *data, sizet sz) ;

バッファ *data* から *sz* バイト分読み出し，このストリームに書き込みます。

writeData void writeData (TStreamable&) ; protected

書き出されるオブジェクトに応じて適切な ***write*** メンバ関数を呼び出すことによって，データをストリームに書き出します。

関連項目 ***TStreamable***，およびストリーム入出力可能なクラスごとに用意されている ***write*** 関数

writePrefix void writePrefix (const TStreamable&) ; protected

ストリームにクラス名のプレフィクスを書き込みます。<< 演算子は，***writeData*** を使用してデータを書き込む際に，この関数を使用し，その前後にプレフィクスおよびサフィクスを書き込みます。プレフィクスとサフィクスは，ストリーム I/O が型に応じて正しく行なわれるようにするためのものです。

関連項目 ***ipstream:readPrefix***

writeString void writeString (const char *str) ;

str をストリームに書き込みます（先頭に，文字列長を表わす 1 バイトが付加されます）。

writeSuffix void writeSuffix (const TStreamable&) ; protected

ストリームにクラス名のサフィクスを書き込みます。<< 演算子は，***writeData*** を使用してデータを書き込む際にこの関数を使用し，その前後にプレフィクスおよびサフィクスを書き込みます。プレフィクスとサフィクスは，ストリーム I/O が型に応じて正しく行なわれるようにするためのものです。

関連項目 ***ipstream:readPrefix***

writeWord void writeWord (ushort us) ;

ワード *us* をストリームに書き込みます。

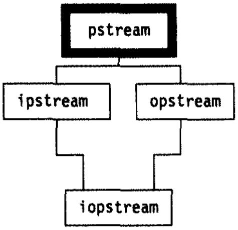
フレンド

operator <<

```
friend opstream& operator << ( opstream& ps , signed char ch ) ;
friend opstream& operator << ( opstream& ps , unsigned char ch ) ;
friend opstream& operator << ( opstream& ps , signed short sh ) ;
friend opstream& operator << ( opstream& ps , unsigned short sh ) ;
friend opstream& operator << ( opstream& ps , signed int i ) ;
friend opstream& operator << ( opstream& ps , unsigned int i ) ;
friend opstream& operator << ( opstream& ps , signed long l ) ;
friend opstream& operator << ( opstream& ps , unsigned long l ) ;
friend opstream& operator << ( opstream& ps , float f ) ;
friend opstream& operator << ( opstream& ps , double d ) ;
friend opstream& operator << ( opstream& ps , long double d ) ;
friend opstream& operator << ( opstream& ps , TStreamable& t ) ;
friend opstream& operator << ( opstream& ps , TStreamable *t ) ;
```

指定された ***ipstream*** オブジェクトに対して，指定された引数の挿入（書き込み）を行ないます。引数に指定されたデータの型によって，どの形式の書き込みが行なわれるかが決まります。

ストリームに関しては，第 8 章，『ストリーム入出力可能なオブジェクト』も参照してください。



pstream は，ストリーム入出力可能なオブジェクトを処理するための基本クラスです。これは，Borland C++ のストリームライブラリにおいて iostream.h で定義されている ***ios*** と同様の機能を果たします。以下のオブジェクトストリームは，Borland C++ 標準の ***iostream*** クラスで使用されているものと同じ ***streambuf*** クラスで機能します。したがって，***iostream*** を基にしてモデムとの間で読み書きを行なうクラスを作成すれば，モデムを介してオブジェクトのやりとりを行なうことも可能になります。

データメンバ

bp streambuf * bp ; protected

ストリームバッファへのポインタ。

state int state ; protected

ios で enum 定義されているフォーマット状態フラグ。現在の状態にアクセスしたい場合には，***rdstate*** を使用してください。

関連項目 ***ios***，***pstream::rdstate***

types static TStreamableTypes *types; protected

このアプリケーションで登録したすべての型を納めている ***TStreamableTypes*** データベースへのポインタ。

関連項目　***TStreamableTypes***, ***pstream::initTypes***

メンバ関数

コンストラクタ　pstream (streambuf * buf) ;

pstream ( ) ;　　**protected**

最初の形式は，指定されたバッファを使用する ***pstream*** を新たに作成し，データメンバ ***bp*** に ***buf*** を設定します。状態は 0 に設定されます。2 番目の形式は，デフォルトのバッファを割り当てます。

デストラクタ　virtual ~pstream ( ) ;

pstream オブジェクトを消去します。

bad　int bad ( ) const ;

エラーが発生した場合にゼロ以外を返します。

clear　void clear (int aState = 0) ;

ストリームの ***state*** を指定した値（デフォルトは 0）に設定します。

eof　int eof ( ) const ;

ストリームの終わりを検出したときにゼロ以外を返します。

error　void error (StreamableError , const TStreamable&) ;

protected

void error (StreamableError) ;

指定されたエラー条件を設定します。***StreamableError*** の定義は次のとおりです。

```
enum StreamableError { peNotRegistered, peInvalidType };
```

fail int fail ( ) const ;

ストリーム処理が失敗した場合にゼロ以外を返します。

good int good ( ) const ;

状態ビットが何も立っていない場合（すなわち，エラーが何も発生していない場合）にゼロ以外を返します。

init void init (streambuf * sbp) ; protected

ストリームを初期化します。***state*** に 0 を，***bp*** に *sbp* を設定します。

initTypes static void initTypes ( ) ;

リンク付けされる ***TStreamableTypes*** オブジェクト * *types* を新たに作成します。***TStreamableClass*** コンストラクタから呼ばれます。

関連項目 ***TStreamableTypes***，***TStreamableClass***

operator void *( ) operator void * ( ) const ;

ポインタを void に変換するキャスト演算子をオーバーロードします。オーバーロードできなかった場合(すなわち，***pstream::fail*** がゼロ以外を返した場合）には 0 を返します。オーバーロードできた場合にはゼロ以外を返します。

関連項目 ***pstream::fail***

operator ! int operator ! ( ) const ;

NOT 演算子をオーバーロードします。***pstream::fail*** が返してきた値を返します。

関連項目 ***pstream::fail***

rdbuf streambuf * rdbuf ( ) const ;

このストリームで割り当てられたバッファへのポインタ ***pb*** を返します。

関連項目　***pstream::pb***

rdstate　int rdstate ( ) const ;

現在の ***state*** の値を返します。

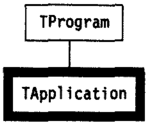
setstate　void setstate (int b) ;　protected

state |= (b & 0xFF) として，データメンバ ***state*** を更新します。

フレンド

クラス ***TStreamableTypes*** は，***pstream*** のフレンドです。

TApplication　APP.H



TApplication は，単に ***TProgram*** を「包み込んだ」形になっており，コンストラクタとデストラクタだけが ***TProgram*** と異なります。Turbo Vision のサブシステム（メモリ，ビデオ，イベント，システムエラー，ヒストリーリストマネージャ）は，すべて静的オブジェクトです。したがって，***main*** に制御が渡る前にコンストラクトされ，***main*** から抜け出た時点ですべて消去されます。

通常アプリケーションを作成する場合には，***TApplication*** から導出する形で作成すれば充分でしょう。それでもサブシステムの初期化やシャットダウンを別の方法で行ないたい場合には，アプリケーションの導出を ***TProgram*** から行ない，Turbo Vision サブシステムの初期化およびシャットダウンを独自のシーケンスで行なうようにすることもできます。

メンバ関数

コンストラクタ　TApplication ( ) ;　protected

TApplication::TApplication は, 以下のような記述となっています。

```
TApplication::TApplication() :
    TProgInit( &TApplication::initStatusLine,
               &TApplication::initMenuBar,
               &TApplication::initDeskTop)
{
   initHistory();
}
```

これは, ***TProgInit*** コンストラクタに ***init*** 関数のポインタを 3 個渡し, デフォルトの ***TApplication*** オブジェクトを作成するという内容になっています。これによって ***TApplication*** オブジェクトには, 全画面ビューが 1 個割り当てられます。また, ***initScreen*** を呼び出し, 画面モードに依存する各種変数の設定を行ないます。画面バッファの割り当ても行ないます。その後, ***initDeskTop***, ***initStatusLine***, ***initMenuBar*** を呼び出し, アプリケーションで使用される基本的な 3 個の Turbo Vision ビューを作成します。デスクトップ, ステータスライン, およびメニューバーの各オブジェクトは, アプリケーションのグループに挿入されます。データメンバ ***state*** は, (***sfVisible*** | ***sfSelected*** | ***sfFocused*** | ***sfModal*** | ***sfExposed***) に設定され, データメンバ ***options*** は, ゼロに設定されます。最後にポインタ ***application*** を (このオブジェクトを指すように) 設定し, また ***initHistory*** が呼び出して, リンク付けされる ***THistory*** オブジェクトの初期化を行ないます。

関連項目　TProgInit::TProgInit, initScreen, initHistory

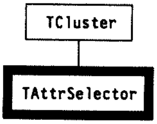
デストラクタ　virtual ~TApplication ( ) ;　protected

アプリケーションオブジェクトを消去します。また, 基本のデストラクタを呼び出してリンクしているオブジェクトを全部消去し, 割

り当てられていたメモリをすべて解放します。デストラクタの記述は以下のとおりです。

```
TApplication::~TApplication
{
   doneHistory();
}
```

TAttrSelector　　　COLORSEL.H



TColorItem, ***TColorGroup***, ***TColorSelector***, ***TAttrSelector***, ***TColorDisplay***, ***TColorGroupList***, ***TColorItemList***, および ***TColorDialog*** は，相互に関係のあるクラスで，ユーザが利用可能なパレットから色の割り当てを選択，変更できるようにするためのビュアーやダイアログボックスを作成する機能を果たします。その際，色の選択や変更を行なうと，その結果が即座に画面に現われます。

TAttrSelector は，ノーマルアトリビュートモード時のアトリビュートとして，ノーマル，アンダーライン，またはリバースを選択するためのクラスタです。

メンバ関数

コンストラクタ　TAttrSelector (const TRect& bounds) ;

TCluster コンストラクタを呼び出し，Normal, Underline, および Reverse のラベルを付けた 3 個のボタンを持つクラスタを作成します。***eventMask*** の ***evBroadcast*** フラグを立てます。

build　static TStreamable * build ( ) ;

ストリーム読み出しを行なう際，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***, ***ipstream::readData***, ***TStreamable***

draw virtual void draw ( ) ;

セレクタクラスタを表示します。

handleEvent virtual void handleEvent (TEvent& event) ;

TCluster::handleEvent を呼び出し，***cmColorSet*** イベントを受け取ると，それにしたがってデータメンバ ***value*** を変更します。必要ならビューの再表示を行ないます。***value*** には，実際には選択された属性に対応するアトリビュートが格納されます。

関連項目 ***TCluster::handleEvent***, ***TCluster::value***

mark virtual Boolean mark (int item) ;

選択されたボタンが *item* 番目だったときに *True* を返します。それ以外の場合には *False* を返します。

movedTo void movedTo (int item) ;

value に *item* 番目の属性をセットします（***press*** と同じ働きをします)。

関連項目 ***TAttrSelector::press***

newColor void newColor ( ) ;

属性が変更された場合に，そのことをグループのオーナーに知らせます。

press virtual void press (int item) ;

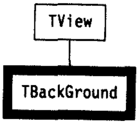
item 番目のボタンを「押下」し，***newColor*** を呼び出します。

関連項目 ***TAttrSelector::newColor***

関連する関数

演算子関数で ***TAttrSelector*** に関係するものもありますが，メンバ関数とはなっていません。詳細については322ページを参照してください。

TBackground　　　　APP.H



TBackground は，一様なパターンをした矩形領域からなる単純なビューです。通常，***TDeskTop*** が所有しています。

データメンバ

pattern　ushort pattern ;　　**protected**

ビューのバックグラウンドに使われる文字（表示コード）。

メンバ関数

コンストラクタ　TBackground (const TRect& bounds, ushort aPattern) ;

TView コンストラクタを呼び出して，指定した *bounds* を持つ ***TBackground*** クラスを作成します。***growMode*** には ***gfGrowHiX*** | ***gfGrowHiY*** が設定され，データメンバ ***pattern*** には *aPattern* が設定されます。

コンストラクタ　TBackground (StreamableInit streamableInit) ;　　protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第8章も参照してください。

関連項目　***TView::TView***，***TBackground::pattern***

build　static TStreamable ＊ build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***，***TStreamable***

draw　virtual void draw ( ) ;

バックグラウンドビューの矩形領域を，現在の ***pattern*** を使ってデフォルトカラーで塗りつぶします。

getPalette　virtual TPalette& getPalette ( ) const ;

デフォルトのバックグラウンドパレット文字列である ***cpBackground*** すなわち "¥x01" を返します。

read　virtual void ＊ read (ipstream& is) ;

入力ストリーム *is* から読み込みを行ないます。

関連項目　***TStreamable***，***ipstream***

write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き出しを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***opstream***

関連する関数

演算子関数で ***TBackground*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TBackgound のパレット

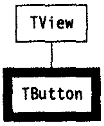
バックグラウンドオブジェクトは，デフォルトパレット ***cpBackground*** を使用し，アプリケーションパレットの第 1 エントリに割り付けます。

TBufListEntry　BUFFERS.H

TBufListEntry は，Turbo Vision が内部的に使用する関数であり，***TVMemMgr*** と連携して，グループ表示を行なうときに使用されるビデオキャッシュバッファの作成と管理を行ないます。メンバはすべてプライベートであり，通常のアプリケーションから直接的に参照されることはまずありません。***TVMemMgr*** は，フレンドクラスであり，グローバル演算子 ***new*** はフレンド関数です。

関連項目　***TGroup::draw***，***TGroup::buffer***，演算子 ***new***

TButton　DIALOGS.H



TButton オブジェクトは，タイトルと影を持つボックスです。押されたときにコマンドを発生します。ボタンは，ハイライトしている文字をキー入力するか，***TAB*** でそのボタンのところに移動して**スペース**を押すか，またはそのボタンがデフォルトになっている場合（そのときにはハイライトしている）には，**リターン**キーを押すか，マウスでそのボタンをクリックすることによって，選択することができます。

ボタンは，選択されたときに 3 次元的な動きをしたかのような印象を与えます。モノクロシステムではボタンはカッコで区切られ，カッコなどの文字を使用して，そのボタンがデフォルトであるとか，選択されているといったことが示されます。

TDialogs で定義されている他のコントロールと同様，***TButton***

も「ターミナル」クラスの 1 種です。どのようなグループにもインサートすることができ，そのグループのメンバ関数をオーバーライドせずに利用できるようになっています。

ボタンを初期化するときには，ボタンに ***TRect***，タイトル文字列，押されたときに発行するコマンド，および unsigned short 型の ***aFlags*** を渡してやります。ボタンにホットキーを定義したい場合には，タイトル文字列の中の 1 文字をチルダ記号（˜）で囲みます。囲まれた文字がホットキーとなります。***aFlags*** 引数は，タイトルを中央に置くか，左詰めにするかを指定します。また ***aFlags*** で，そのボタンをデフォルトとするかどうか（つまり**リターン**キーで選択されるようにするかどうか）も指定します。

ウィンドウやダイアログで，デフォルトボタンとなれるのは 1 個だけです。グループ内で同列のボタンは，***evBroadcast*** メッセージを使ってデフォルト状態を獲得したり手放したりします。ボタンは，メンバ関数である ***setState***，***enableCommand***，および ***disableCommand*** を使用して，有効にしたり無効にしたりすることができます。

データメンバ

amDefault Boolean amDefault ;

True の場合には，そのボタンがデフォルトです(この場合，**リターン**キーが押されると選択されます)。*False* の場合には，そのボタンは「ノーマル」です。

関連項目 ***bfXXXX*** ボタンフラグ定数

command ushort command ;

このボタンが押されたときに発生するイベントを表わすコマンドワード。

関連項目 ***TButton*** のコンストラクタ

flags uchar flags ;

flags はビットマップのデータメンバであり，ボタンのテキストを左詰めにするか中央揃えにするかを指定します。個々のフラグについては，第16章の「**bfXXXX ボタンフラグ定数**」で説明しています。

関連項目 ***TButton::draw***，***bfXXXX*** ボタンフラグ定数

title const char ＊ title ;

ボタンのラベルを示す文字列。

メンバ関数

コンストラクタ TButton (const TRect& bounds, const char ＊ aTitle, ushort aCommand, ushort aFlags) ;

コンストラクタ ***TView*** (*bounds*) を呼び出して，指定したサイズの ***TButton*** クラスを作成します。*aTitle* 文字列が ***title*** に設定されます。*aFlags* の ***bfDefault*** ビットが立っていれば，このボタンはデフォルトボタンとなりハイライトします。***bfLeftJust*** ビットがクリアされていれば，ボタンのタイトルは中央揃えになります。逆に ***bfLeftJust*** が立っていれば左詰めになります。

データメンバ ***options*** は，(***ofSelectable*** | ***ofFirstClick*** | ***ofPreProcess*** | ***ofPostProcess***) に設定されるため，デフォルトでこれらのイベントに応答することになります。***eventMask*** は，***evBroadcast*** に設定されます。*aCommand* の値によって，データメンバ ***state*** の内容が設定されます。たとえば，指定された *aCommand* が有効となっていない場合には ***sfDisabled*** がデータメンバ ***state*** に設定されます。

コンストラクタ TButton (StreamableInit streamableInit) ; **protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンス

トラクタを呼び出すことで行なうことができます。これについては第8章も参照してください。

関連項目　***TView*** コンストラクタ，***bfXXXX*** ボタンフラグ定数

デストラクタ　~TButton ( ) ;

ボタンの ***title*** に割り当てられていたメモリを解放し，***~TView*** を使用してそのビューを消去します。

関連項目　***~TView***

build　static TStreamable * build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

draw　virtual void draw ( ) ;

TButton::drawState(*False*) を呼び出し，ボタンの表示を行ないます。

関連項目　***TButton::drawState***

drawState　void drawState (Boolean down) ;

down が ***True*** ならボタンを「押された」状態にして表示し，***down*** が ***False*** なら「押されていない」状態にして表示します。現在の状態（ノーマル，デフォルト，無効）にしたがって，それに合わせたパレットが使用されます。ボタンのラベルの位置は，データメンバ ***flags*** の ***bfLeftJust*** ビットにしたがって決められます。

関連項目　***TButton::draw***，***TButton::drawTitle***，***bfXXXX*** フラグ

getPalette　virtual TPalette& getPalette ( ) const ;

ボタンのデフォルトのパレット文字列 ***cpButton***，すなわち "¥x0A¥x0B¥x0C¥x0D¥x0E¥x0E¥x0E¥x0F" を返します。

handleEvent　virtual void handleEvent (TEvent& event) ;

そのボタン上でマウスがクリックされるか，ホットキーが押されるか，あるいはデフォルトボタンが ***cmDefault*** ブロードキャストを受信するかのいずれかの場合に処理を行ないます。ボタンが押された場合には，***TView::putEvent*** によってコマンドイベントが発行されます。その際，***event::command*** にはそのボタンのデータメンバ ***TButton::command*** の内容が入り，***event::infoPtr*** はこのボタンを指すように設定されます。

ボタンはまた，必要に応じてデフォルトボタンになったり，ならなかったりするためにあるブロードキャストコマンド ***cmGrab-Default***，***cmReleaseDefault*** や，ボタンの発行したコマンドが有効にされたり，無効にされたりしていないかをボタンに確認させるためのブロードキャストコマンド ***cmCommandSetChanged*** についても，対処するようになっています。

関連項目 ***TView::handleEvent***

makeDefault void makeDefault (Boolean enable) ;

enable を *True* にすればこのボタンをデフォルトにし，***enable*** を *False* にすれば，このボタンにデフォルトの権利を手放させます。これは，通常ダイアログボックスで ***TAB*** キーを押した場合に起こります。実際にボタンの処理を行なうことなく，ステータスが変わります。これによって，デフォルトとなったボタンは，**リターン**キーを入力することで「押される」ようになります。***ma-keDefault*** は，そのボタンがすでにデフォルトボタンであった場合には何もしません。そのボタンがデフォルトボタンでなかった場合には，ボタンのオーナーに対してそのボタンのデフォルトステータスが変わったことが伝えられます。*enable* が *True* の場合には，***cmGrabDefault*** コマンドがブロードキャストされます。*enable* が *False* の場合には，***cmReleaseDefault*** がブロードキャストされます。ステータスが変わった場合には，ボタンの再表示が行なわれます。

関連項目 ***TButton::amDefault***，***TButton::press***，***bfDefault***

press void press ( ) ;

press は，ボタン押下イベントが発生したことを表わすメッセージをブロードキャストします。マウスクリックによる「押下」が検出された場合，および**リターン**キーの入力によってデフォルトボタンが「押された」場合に，***handleEvent*** が内部的に使用します。

read virtual void * read (ipstream& is) ;

入力ストリーム *is* から読み出しを行ないます。

関連項目 ***TStreamableClass***，***TStreamable***，***ipstream***

setState void setState (ushort aState, Boolean enable) ;

TView::setState (*aState, enable*) を呼び出し，***sfSelected*** か ***sfActive*** にされた場合には，そのボタンの表示(***drawView***)を行ないます。ボタンがフォーカスを受け取った場合（すなわち，*aState* が ***sfFocused*** の場合）には，***makeDefault*** (*enable*) を呼び出して，デフォルトボタンからデフォルトの権利を奪い取ったり，デフォルトの権利を手放したりします。

関連項目 ***TView::setState***，***TButton::makeDefault***

write virtual void write (opstream& os) ;

出力ストリーム *os* に書き込みを行ないます。

関連項目 ***TStreamableClass***，***TStreamable***，***opstream***

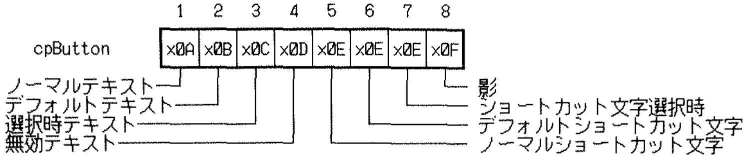
関連する関数

演算子関数で ***TButton*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

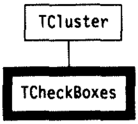
TButton のパレット

ボタンオブジェクトは，デフォルトパレット ***cpButton*** を使用し，

cpDialog パレットのエントリ 10〜15 に割り付けます。



TCheckBoxes　　DIALOGS.H



TCheckBoxes は，1〜16 個までのコントロールで構成される特殊なクラスタです。ラジオボタンの場合と異なり，チェックボックスは何個でもマークすることができます。したがって，そのグループでデフォルトのチェックボックスというのはありません。マウスクリック，カーソルの移動，**GRPH-文字**によるショートカットによってマーキングを行なうことができます。チェックボックスはハイライト表示にすることができ，**スペース**キーでオン/オフを切り替えることができます。選択されたときにはボックス内に X が表示されます。通常，アプリケーション側からはチェックボックスの状態を調べて，ユーザがどのオプションを選択したのかを判断します。チェックボックスクラスタは，通常 ***TLabel*** オブジェクトとリンクして使用されます。

データメンバ

TCluster から継承した ***value***, ***sel***, およびいくつかの文字列以外には public のデータメンバはありません。***ushort*** である ***value*** は，16 個のビット集合(0〜15)とみなされ，第 *item* 番目のビットが 1 であれば *item* 番目のチェックボックスにマークがつけられ

ているということを意味します。

メンバ関数

コンストラクタ　TCheckBoxes (const TRect& bounds, TSItem * aStrings) ;

TCluster (*bounds*, *aStrings*) コンストラクタおよび ***TView*** (*bounds*) コンストラクタを呼び出し，指定した *bounds* を持つ ***TCheckBox*** オブジェクトを作成します。****strings*** のデータメンバである ***TStringCollection*** には，***TSItem*** オブジェクトのリンクリストを表わす引数 *aStrings* が渡されます。データメンバ ***sel*** および ***value*** には，0 が設定されます。***options***には，(***ofSelectable*** | ***ofFirstClick*** | ***ofPreProcess*** | ***ofPostProcess***) が設定されます。

コンストラクタ　TCheckBoxes (StreamableInit streamableInit);　protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TCluster::TCluster***，***TSItem***，***ofXXXX*** フラグ

デストラクタ　***TCheckBoxes*** は，基本クラスタのデストラクタである ***~TCluster*** を使用して文字列を消去します。その後，***TView*** デストラクタがこのビューを廃棄します。

関連項目　***TCluster::~TCluster***

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

draw virtual void draw ( ) ;

継承した ***TCluster::drawBox*** メンバ関数を呼び出して ***TCheckBoxes*** オブジェクトを表示します。デフォルトのチェックボックスは，選択されていない場合が［ ］で，選択されている場合が［X］です。

ビューの幅に余裕がある場合には，チェックボックスの桁数をふやすこともできます。

関連項目 ***TCluster::drawBox***，***TCluster::column***

mark virtual Boolean mark (int item) ;

value の第 *item* 番目のビットが立っていれば，つまり第 *item* 番目のチェックボックスがマークされていれば *True* を返します。これらのビットには，最初から決められた意味というものはありません。***mark*** や ***press***，あるいはチェックボックスのその他のメンバ関数は，自由にオーバーライドして ***value*** に独自の意味を持たせてかまいません。デフォルトでは，項目には 0～15 の番号がつけられており，***value*** の各ビットがチェックボックスの状態（オンかオフか）を表わすことになっています。

関連項目 ***TCheckBoxes::press***，***TCluster::value***

press virtual void press (int item) ;

value の第 *item* 番目のビットを反転させます。これらのビットには，最初から決められた意味というものはありません。***mark*** や ***press***，あるいはチェックボックスのその他のメンバ関数は，自由にオーバーライドして ***value*** に独自の意味を持たせてかまいません。デフォルトでは，項目に 0～15 の番号が付けられています。

関連項目 ***TCheckBoxes::mark***，***TCluster::value***

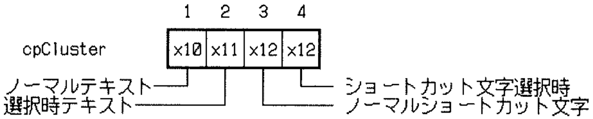
関連する関数

演算子関数で ***TCheckBoxes*** に関係するものもありますが，メンバ

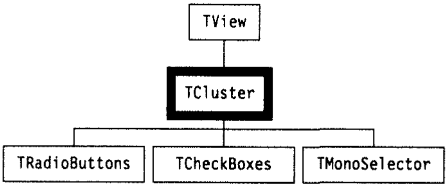
関数とはなっていません。詳細については 322 ページを参照してください。

パレット

デフォルトでは，チェックボックスオブジェクトはすべてのクラスタオブジェクトのデフォルトパレットである ***cpCluster*** を使用します。



TCluster DIALOGS.H



クラスタとは，同じ応答の仕方をする複数個のコントロールからなるグループのことです。***TCluster*** は抽象クラスであり，***TCluster*** から ***TRadioButtons*** や ***TCheckBoxes*** や ***TAttrSelector*** などの便利なコントロールグループが導出されます。クラスタの各コントロールには，ふつう ***TLabel*** オブジェクトがリンクされ，コントロールの横にあってコントロールの意味を示しているラベルを選択することによってコントロールを選択できるようになっています。

ボタンがコマンドを発生するために使用され，入力行が文字列を編集するために使用されるのに対してクラスタは，***ushort*** 型のデータメンバである ***value*** の各ビット値を反転するために使用されます。***TCluster*** の標準的な下位クラスである ***TCheckBoxes*** と

TRadioButtons は，***value*** を変更するアルゴリズムが異なります。***TCheckBoxes*** では，あるビットを単に反転させるだけですが，***TRadioButtons*** では有効にされるビットを反転するとともに，それまで選択されていたビットをクリアします。どちらのクラスもほとんどの機能を ***TCluster*** から継承しています。

データメンバ

sel int sel ;

クラスタの中で現在選択されている項目。

strings TStringCollection ＊ strings ;

クラスタ内の項目のリスト。

value ushort value ;

コントロールの現在値。このデータメンバの意味は，実際には ***TCluster*** から導出されたクラスにおいて定義されたメンバ関数によって決まります。たとえば ***TCheckBoxes*** の場合には，***value*** の16 個のビットの中の各ビットは16 個のチェックボックスそれぞれの状態（オンかオフ）を表わすものとされます。一方 ***TRadioButtons*** の場合，オンになれるラジオボタンは常に 1 個なので，***value*** によって最大 65,536 個までのボタンの状態を表わすことができます。(***ushort*** は，現在のところ，0〜65,535 の値を表わす 16 ビットの unsigned short として typedef されています。)

メンバ関数

コンストラクタ TCluster (const TRect& bounds, TSItem ＊ aStrings) ;

TView (*bounds*) を呼び出し，指定した *bounds* を持つ ***TCluster*** オブジェクトを作成します。データメンバの ***value*** と ***sel*** はクリ

アされます。***strings** のデータメンバである **TStringCollection** には，***TSItem*** オブジェクトのリンクリストを表わす引数 *aStrings* が渡されます。

コンストラクタ　TCluster (StreamableInit streamableInit) ;　protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TSItem***, ***TView::TView***

デストラクタ　~TCluster ( ) ;

クラスタの文字列コレクションを消去してから，***~TView*** を呼び出してビューを廃棄します。

関連項目　***TView::~TView***

build　virtual Tstreamable * build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass, ipstream::readData***

dataSize　virtual ushort dataSize ( ) ;

value のサイズを返します。下位クラスにおいて ***value*** の意味を変更するか，データメンバをふやした場合には，***getData*** および ***setData*** と連携して機能するようにオーバーライドする必要があります。

関連項目　***TCluster::getData***, ***TCluster::setData***

drawBox　void drawBox (const char * icon, char marker) ;

導出クラスのメンバ関数 ***draw*** から呼び出され，クラスタ内の各項目の文字列の前（左横）にボックスを表示します。アイコンには，

5 文字の文字列が使われます（チェックボックスの場合は " [ ] ", ラジオボタンの場合は "( )"）。*marker* は，そのボックスがマークされたことを示すための文字です（チェックボックスには "X" が，ラジオボタンには "O" が使用されます）。

関連項目 ***TCheckBoxes::draw***, ***TRadioButtons::draw***

getData virtual void getData (void * rec) ;

データメンバ ***value*** の値を指定されたレコードに書き込み，***drawView*** を呼び出します。導出クラスにおいて，データメンバ ***value*** の意味を変更した場合には，***dataSize*** および ***setData*** と連携して機能するようにオーバーライドする必要があります。

関連項目 ***TCluster::dataSize***, ***TCluster::setData***, ***TView::drawView***

getHelpCtx ushort getHelpCtx ( ) ;

(***sel*** + ***helpCtx***) を返します。これによって，クラスタ内の各項目ごとに別々のヘルプコンテキストを持たせることができます。この関数を使用して，***helpCtx***＋クラスタの項目数－1 までのヘルプコンテキストを使用しないようにしてください。

getPalette virtual TPalette& getPalette ( ) const ;

デフォルトパレットである ***cpCluster***, すなわち "¥x10¥x11¥x12¥x12¥x12¥" へのポインタを返します。

handleEvent virtual void handleEvent (TEvent& event) const ;

TView::handleEvent を呼び出した後，このクラスタで処理すべきすべてのマウスイベントおよびキーボードイベントの処理を行ないます。コントロールは，マウスクリックかカーソル移動キー（**スペース**キーも含めて）によって選択されます。選択されたコントロールを表示するために，クラスタの再表示を行ないます。

関連項目 ***TView::handleEvent***

mark virtual Boolean mark (int item) ;

どの項目がマークされているかを知るために，***draw*** が呼び出します。デフォルトの ***TCluster::mark*** は，*False* を返します。クラスタ内の *item* 番目のコントロールがマークされている場合には *True* を返し，そうでない場合には *False* を返すように ***mark*** をオーバーライドする必要があります。

関連項目 ***TCheckBoxes::mark***，***TRadioButtons::mark***

movedTo virtual void movedTo (int item) ;

handleEvent が，選択バーを *item* 番目のコントロールに移動させるために呼び出します。

press virtual void press (int item) ;

クラスタ内の *item* 番目のコントロールが，マウスクリックかキーボードのイベントによって押されたときに ***handleEvent*** が呼び出します。これは抽象メンバ関数であり，オーバーライドする必要があります。

関連項目 ***TCheckBoxes::press***，***TRadioButtons::press***

read virtual void *read (ipstream& is) ;

入力ストリーム *is* から読み出しを行ないます。

関連項目 TStreamableClass，TStreamable，ipstream

setData virtual void setData (void *rec) ;

指定されたレコードからデータメンバ ***value*** を読み出し，***drawView*** を呼び出します。導出したクラスでデータメンバを増やす場合には，***dataSize*** および ***getData*** と連携して機能するようにオーバーライドする必要があります。

関連項目 ***TCluster::dataSize***，***TCluster::getData***，***TView::drawView***

setState virtual void setState (ushort aState, Boolean enable) ;

TView::setState を呼び出し，*aState* が ***sfSelected*** であれば

drawView を呼び出します。

関連項目　***TView::setState***, ***TView::drawView***

write　virtual void write (opstream& os) ;

出力ストリーム *os* に書き込みを行ないます。

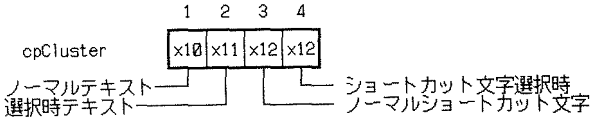
関連項目　***TStreamableClass***, ***TStreamable***, ***opstream***

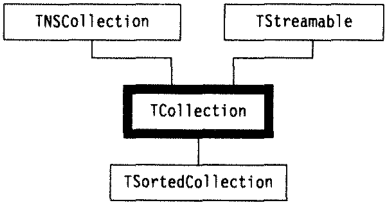
関連する関数

演算子関数で ***TCluster*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

TCluster オブジェクトは，すべてのクラスタオブジェクトでデフォルトとなっているパレット ***cpCluster*** を使用し，ダイアログボックスの標準パレットのエントリ 16〜18 に割り付けます。





TCollection は，他のオブジェクトも含めた任意の要素からなるストリーム入出力可能なコレクションを作成するためのものです。通常，実際に使用されるストリーム入出力可能なコレクションクラスを導出する際の基本クラスとして利用されます。***TNSCollection***（ストリーム入出力可能でないコレクションクラス）が ***TCollection*** の仮想基本クラスとなり，コレクションに対して要素を追加したり，アクセスしたり，要素の削除を行なったりします。***TStreamable*** は，***TCollection*** にストリームを作成したり，ストリームに名前を付けたりする機能を付与します。いくつかのフレンド関数およびオーバーロード演算子 >>, << によって，ストリームとの間でコレクションの読み書きがサポートされます。

コレクションは，従来の配列や集合やリストよりも一般的な概念です。***TCollection*** オブジェクトは，実行時に動的に自分自身のサイズを決定することができ，***TSortedCollection*** や ***TStringCollection***, ***TResourceCollection*** などの特殊化されたクラスを導出するときの基本となります。***TCollection*** は，***TNSCollection*** から，要素の追加や削除を行なうメンバ関数の他に，ある関数をコレクション内の１個１個の要素に対して呼び出していくという**繰り返し処理**ルーチンも継承します。

データメンバ

name static const char *const name ;

コレクションクラスの名前 "TCollection"。ストリームマネージャが内部的に使用します。

関連項目　*TSortedCollection::TSortedCollectionName*

メンバ関数

コンストラクタ　TCollection (ccIndex aLimit, ccIndex aDelta) ;

limit に *aLimit* を，***delta*** に *aDelta* を設定してコレクションを作成します。要素数は *aLimit* に初期設定されますが，メモリを使い切るか，要素数が ***maxCollectionSize*** に達するまで *aDelta* 個単位でコレクションを拡張することができます。

コンストラクタ　TCollection (StreamableInit streamableInit) ;　**protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TNSCollection::TNSCollection***, ***TNSCollection::limit***, ***TNSCollection::delta***

read　void read (ipstream& is) ;　**protected**

入力ストリーム *is* からコレクションを読み出し，そのコレクションとリンクしている ***TCollection*** オブジェクトに格納します。

関連項目　ipstream

readItem　virtual void * readItem (ipstream& is) = 0 ;　**private**

これは純粋仮想関数であり，***ipstream*** から型の対応をとりながら要素を読み出してくる関数として導出クラスで定義する必要があります。この読み出しは，通常，導出クラスの各データメンバに対し

て一連の >> 演算を施すことによって行なわれます。

write　void write (opstream& os) ;　　protected

出力ストリーム *os* にリンク付けされたコレクションを書き込みます。

関連項目　***opstream***

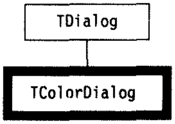
writeItem　virtual void writeItem (void * item, opstream& os) = 0 ;　　private

これは純粋仮想関数であり，***opstream*** に要素を書き込む関数として導出クラスで定義する必要があります。この書き込みは通常，導出クラスの各データメンバに対して一連の << 演算を施すことによって行なわれます。

関連する関数

演算子関数で ***TCollection*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TColorDialog　　COLORSEL.H



TColorItem, ***TColorGroup***, ***TColorSelector***, ***TAttrSelector***, ***TColorDisplay***, ***TColorGroupList***, ***TColorItemList***, および ***TColorDialog*** は相互に関係のあるクラスであり，ユーザが利用可能なパレットから色の割り当てを選択，変更できるようにするためのビュアーやダイアログボックスを作成する機能を果たします。その際，色の選択や変更を行なうと，その結果が即座に

画面に現われます。

TColorDialog は，Colors と呼ばれる用途の特殊化されたスクロール可能なダイアログボックスであり，パレットの選択をいろいろと確かめてみてから実際の選択を行なえるようにします。***TColorDialog*** は，上述した多数のクラスを使用します。それらの各クラスについての説明も参照してください。

データメンバ

attrLabel TLabel *attrLabel ;

ノーマルアトリビュート用のアトリビュートのラベル。

関連項目 ***TLabel***

attrSel TAttrSelector *attrSel ;

ノーマルアトリビュート用の色のラベル。

関連項目 ***TLabel***

color Sel TColorSelector *colorSel ;

ノーマルアトリビュート用の色のセレクタ。

関連項目 ***TColorSelector***

bakLabel TLabel *bakLabel ;

バックグラウンド色のラベル。

関連項目 ***TLabel***

bakSel TColorSelector *bakSel ;

バックグラウンド色のセレクタ。

関連項目 ***TColorSelector***

display TColorDisplay *display ;

このダイアログボックスのカラー表示を行なうオブジェクト。

関連項目　***TColorDisplay***

forLabel　TLabel *forLabel ;

文字色のラベル。

関連項目　***TLabel***

forSel　TColorSelector *forSel ;

文字色のセレクタ。

関連項目　***TColorSelector***

groups　TColorGroupList *groups ;

このダイアログボックス用のカラーグループ。

関連項目　***TColorGroupList***

pal　uchar *pal ;

現在のパレット選択。

関連項目　***TPalette***

メンバ関数

コンストラクタ　TColorDialog (uchar *aPalette, TColorGroup *aGroups) ;
TColorDialog (StreamableInit) ;　**protected**

1番目の形式は, ***TDialog*** コンストラクタおよび ***TScrollBar*** コンストラクタを呼び出し, スクロールバーを 2 本備え, "Colors" というタイトルを持つ固定のフレーム付きウィンドウを作成します。データメンバ ***pal*** に ***aPalette*** が設定されます。引数 ***aGroups*** を指定することによって, "~G~roup" というラベルの付いた ***TColorGroup*** オブジェクトが作成され挿入されます。***aGroups*** の各項目は, ***TColorItemsList*** オブジェクトを初期化します。

TColorItemsList オブジェクトも, "~I~tem" というラベルを付けてダイアログボックスに挿入されます。

また, 次のことを行ないます。拡張アトリビュートモード時のための文字色, バックグランド色, ノーマルモード時の色のセレクタを作成, 挿入し, それぞれ "~F~oreground", "~B~ackground" , "~C~olor" というラベルを付けます。ノーマルモード時のアトリビュートのための ***TAttrSelector*** オブジェクトを作成して挿入します。文字列"Text"を現在のテキスト色で表示します。"O~K~" および " Cancel" ボタンが挿入されます。また, すべての項目のうち, 現在のモードに必要な項目が表示されます。

build static TStreamable * build ( ) ;

ストリーム読み出しを行なうときに, オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***, ***ipstream::readData***

dataSize virtual ushort dataSize ( ) ;

デフォルトで, 現在のパレットのサイズを返します。

関連項目 ***TPalette***

getData virtual void getData(void *rec) ;

pal を ***dataSize*** バイト分 *rec* にコピーします。

handleEvent virtual void handleEvent (TEvent& event) ;

TDialog::handleEvent を呼び出し, ***cmNewColorIndex*** のブロードキャストイベントが発行されていた場合には, 再表示を行ないます。

関連項目 ***TDialog::handleEvent***

read virtual void *read (ipstream& is) ;

入力ストリーム *is* から読み出しを行ないます。

関連項目 ***TStreamableClass***, ***TStreamable***, ***ipstream***

setData virtual void setData (void *rec) ;

getData と逆の機能。*rec* を ***pal*** にコピーすることにより，セーブしてあった色選択を復旧します。

write virtual void write (opstream& os) ;

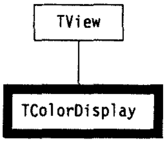
出力ストリーム *os* への書き込みを行ないます。

関連項目 ***TStreamableClass***, ***TStreamable***, ***opstream***

関連する関数

演算子関数で ***TColorDialog*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TColorDisplay COLORSEL.H



TColorItem, ***TColorGroup***, ***TColorSelector***, ***TAttr-Selector***, ***TColorDisplay***, ***TColorGroupList***, ***TColorItem-List***, および ***TColorDialog*** は相互に関係のあるクラスであり，ユーザが画面属性を選択・変更したり，利用可能なパレットから色の割り当てを選択・変更できるようにするためのビュアーやダイアログボックスを作成する機能を果たします。その際色の選択や変更を行なうと，その結果が即座に画面に現われます。

TColorDisplay は，ユーザがパレットを選択する際に，色見本としてテキストを表示するビューです。

データメンバ

color uchar *color ;

テキスト表示に使用される現在の色。

text const char *text ;

表示されるテキスト文字列。

メンバ関数

コンストラクタ TColorDisplay (const TRect& bounds, const char *aText) ;

TView (***bounds***) で得られたサイズでビューを作成し，***text*** を引数 ***aText*** に設定します。

デストラクタ virtual ~TColorDisplay ( ) ;

作成したビューおよび ***text*** 文字列を消去します。

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

draw virtual void draw ( ) ;

設定してあるテキストおよび現在の色を使ってビューを表示します。

handleEvent virtual void handleEvent (TEvent& event) ;

TView::handleEvent を呼び出し，***cmColorBackgroundChanged***，***cmColorForegroundChanged***，***cmNormalColorChanged***，***cmAttrChanged*** のいずれかのブロードキャストイベントが

発行されていた場合に，必要に応じてビューの再表示を行ないます。

関連項目　***TView::handleEvent***

read　virtual void ＊ read (ipstream& is) ;

入力ストリーム *is* から読み出しを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream***

setColor　virtual void setColor (uchar ＊ aColor) ;

color に *aColor* を設定し，色に変更があったことをオーナーのグループにブロードキャストした後，***drawView*** を呼び出します。

関連項目　***TView::drawView***

write　virtual void write (opstream& os) ;

出力ストリーム *os* へ書き込みを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***opstream***

関連する関数

演算子関数で ***TColorDisplay*** に関係するものもありますが，メンバ関数とはなっていません。詳細については322ページを参照してください。

TColorGroup　　COLORSEL.H

TColorItem，***TColorGroup***，***TColorSelector***，***TAttrSelector***，***TColorDisplay***，***TColorGroupList***，***TColorItemList***，および ***TColorDialog*** は相互に関係のあるクラスであり，ユーザが利用可能なパレットから色の割り当てを選択，変更できるよ

うにするためのビュアーやダイアログボックスを作成する機能を果たします。その際，色の選択や変更を行なうと，その結果が即座に画面に現われます。

TColorGroup クラスは，***TColorItem*** オブジェクトをリンクリストにつないだグループを定義します。色グループのメンバは，色の名前とそれにリンクさせた色コードとからなります。

データメンバ

items TColorItem *items ;

この色グループにリンクさせた色項目からなる連想リスト（associated list ）へのポインタ。

name const char *name ;

色グループの名前。

next TColorGroup *next ;

次の色グループへのポインタ。次の色グループがない場合には 0。

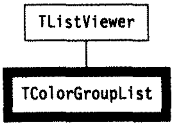
メンバ関数

コンストラクタ

```
TColorGroup ( const char *nm, TColorItem *itm,
              TColorGroup *nxt = 0 ) ;
```

引数に指定された各値を持つ色グループを作成する。

TColorGroupList COLORSEL.H



TColorItem, ***TColorGroup***, ***TColorSelector***, ***TAttr-Selector***, ***TColorDisplay***, ***TColorGroupList***, ***TColorItem-List***, および ***TColorDialog*** は相互に関係のあるクラスであり，ユーザが利用可能なパレットから色の割り当てを選択，変更できるようにするためのビュアーやダイアログボックスを作成する機能を果たします。その際，色の選択や変更を行なうとその結果が即座に画面に現われます。

TColorGroupList は，***TListViewer*** を特殊用途化した導出クラスであり，色グループの名前をスクロール可能なリストにして表示します。色グループは，通常の方法（マウスまたはキーボードを使用する方法）を使用して選択することができます。***TColorGroupList*** は，***TListViewer*** イベントハンドラをそのまま使用します。

データメンバ

groups　TColorGroup * groups ;

このリストビュアーに使用する色グループ。

メンバ関数

コンストラクタ　TColorGroupList (const TRect& bounds,
TScrollBar *aSc-rollBar, TColorGroup *aGroups) ;

TListViewer (*bounds*, 1, 0, *aScrollBar*) を呼び出して，縦方向のスクロールバーのみを備えた1段のリストビュアーを作成します。その後，***groups*** に *aGroups* を設定します。

デストラクタ　virtual ~TColorGroupList ( ) ;

作成したリストビュアー，およびそれにリンクさせていたすべてのグループとその項目を消去します。

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***, ***ipstream::readData***

focusItem virtual void focusItem (int item) ;

TListViewer::focusItem(*item*) を呼び出して指定された *item* を選択した後，***cmNewColorItem*** イベントをブロードキャストします。

関連項目 ***TListViewer::focusItem***

getText virtual void getText (char *dest, int item, int maxLen) ;

item に対応するグループ名を *dest* にコピーします。

read virtual void * read (ipstream& is) ;

入力ストリーム *is* から読み出しを行ないます。

関連項目 ***TStreamableClass***, ***TStreamable***, ***ipstream***

write virtual void write (opstream& os) ;

出力ストリーム *os* に書き込みを行ないます。

関連項目 ***TStreamableClass***, ***TStreamable***, ***opstream***

関連する関数

演算子関数で ***TColorGroupList*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TColorItem　　COLORSEL.H

TColorItem, ***TColorGroup***, ***TColorSelector***, ***TAttrSelector***, ***TColorDisplay***, ***TColorGroupList***, ***TColorItemList***, および ***TColorDialog*** は相互に関係のあるクラスであり，ユーザが利用可能なパレットから色の割り当てを選択，変更できるようにするためのビュアーやダイアログボックスを作成する機能を果たします。その際，色の選択や変更を行なうと，その結果が即座に画面に現われます。

TColorItem は，色の名前とインデックスからなるリンクリストを定義します。

データメンバ

index　uchar index;

項目の色のインデックス。

name　const char *name;

色項目の名前。

next　TColorItem *next;

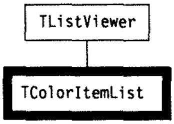
次の色項目へのリンク。次の項目がない場合には 0。

メンバ関数

コンストラクタ TColorItem (const char *nm, uchar idx,
TColorItem *nxt = 0) ;

name に *nm* を設定して，色項目オブジェクトを作成します。***index*** には *idx* が設定され，***next*** にはデフォルトで 0 が設定されます。

TColorItemList COLORSEL.H



TColorItem, ***TColorGroup***, ***TColorSelector***, ***TAttrSelector***, ***TColorDisplay***, ***TColorGroupList***, ***TColorItemList***, および ***TColorDialog*** は相互に関係のあるクラスであり，ユーザが利用可能なパレットから色の割り当てを選択，変更できるようにするためのビュアーやダイアログボックスを作成する機能を果たします。その際，色の選択や変更を行なうと，その結果が即座に画面に現われます。

TColorItemList は，***TColorGroupList*** よりも単純な導出クラスであり，色のグループではなく，単一の色項目を確かめたり選択したりするためのものです。***TColorItemList*** は，***TColorGroupList*** と同様，***TListViewer*** を特殊用途化した導出クラスです。色項目は，通常の方法（マウスまたはキーボードによる方法）で選択することができます。***TColorGroupList*** の場合とは異なり，***TColorItemList*** では，***TListViewer*** イベントハンドラをオーバーライドしています。

データメンバ

items TColorItem *items ;

このビュアーで使用される色項目のリスト。

メンバ関数

コンストラクタ TColorItemList (const TRect& bounds, TScrollBar * aScrollBar, TColorItem *aItems);

TListViewer (*bounds*, 1, 0, *aScrollBar*) を呼び出して，縦方向のスクロールバーのみを備えた1段のリストビュアーを作成します。その後，***items*** に ***aItems*** を，***range*** に項目数を設定します。

コンストラクタ TColorItemList (StreamableInit) ; **protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第8章も参照してください。

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***, ***ipstream::readData***, ***TStreamable***

focusItem virtual void focusItem (int item) ;

TListViewer::focusItem (*item*) を呼び出し，指定された *item*

を選択して，***cmNewColorIndex*** イベントをブロードキャストします。

関連項目　***TListViewer::focusItem***

getText　virtual void getText (char * dest, int item, int maxLen) ;

item に対応する項目の ***name*** を *dest* にコピーします。

handleEvent　virtual void handleEvent (TEvent& event) ;

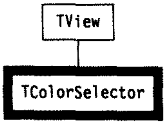
TListViewer::handleEvent を呼び出します。イベントが ***cmNewColorItem*** であった場合には，しかるべき項目のところにフォーカスが行き，ビュアーの再表示が行なわれます。

関連項目　***TListViewer::handleEvent***

関連する関数

演算子関数で ***TColorItemList*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TColorSelector　COLORSEL.H



TColorItem, ***TColorGroup***, ***TColorSelector***, ***TAttrSelector***, ***TColorDisplay***, ***TColorGroupList***, ***TColorItemList***, および ***TColorDialog*** は相互に関係のあるクラスであり，ユーザが利用可能なパレットから色の割り当てを選択，変更できるようにするためのビュアーやダイアログボックスを作成する機能を果たします。その際，色の選択や変更を行なうと，その結果が即座に画面に現われます。

TColorSelector は，利用可能な色の選択肢を表示するビューです。

データメンバ

color uchar color ;

現在選択されている色が格納されています。

selType ColorSel selType ;

選択する色の一覧表の属性（文字色用かバックグラウンド色用かノーマルモード時の色用か）を示します。***ColorSel*** は，enum を使って以下のように定義されています。

```
enum ColorSel { csBackground, csForeground, csNormalColor };
```

メンバ関数

コンストラクタ TColorSelector (const TRect& bounds, ColorSel sSelType) ;

TView (*bounds*) を呼び出して，指定された *bounds* のビューを作成します。***ofSelectable***，***ofFirstClick***，および ***ofFramed*** の各オプションを設定します。また，***eventMask*** に ***evBroadcast*** を，***selType*** に ***aSelType*** を，***color*** に 0 を設定します。

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***，***ipstream::readData***

draw virtual void draw ( ) ;

色セレクタを表示します。

handleEvent virtual void handleEvent (TEvent& event) ;

マウスイベントおよびキーイベントの処理を行ないます。色の選択を行なう場合，その色のインジケータのところをクリックするか，カーソル移動キーを使ってカーソルをその色のところに位置づけて，選択することができます。変更があった場合，必要に応じて ***drawView*** が呼び出されます。

read virtual void ＊ read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***，***TStreamable***，***ipstream***

write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目 ***TStreamableClass***，***TStreamable***，***opstream***

関連する関数

演算子関数で ***TColorSelector*** に関係するものもありますが，メンバ関数とはなっていません。詳細については322ページを参照してください。

TCommandSet VIEWS.H

TCommandSet はビューとならないクラスであり，コマンドセットの処理を行ないます。メンバ関数には，コマンドを有効にしたり，無効にしたり，指定されたコマンドが存在するかどうかを検査したりするものがあります。いくつかの演算子をオーバーロードし，等しいかどうかの検査などを自然な形で行なうことができるように

します。コマンドは，0〜255 の整数として扱われます。

メンバ関数

コンストラクタ

TCommandSet ( );
TCommandSet (const TCommandSet& commands);

1番目の形式の場合，コマンドセットの作成とクリアを行ないます。2番目の形式の場合，コマンドセットを作成し，引数 *commands* で初期化します。

disableCmd

void disableCmd (int cmd);
void disableCmd (const TCommandSet& tc);

この関数を呼び出したコマンドセットから，*cmd*（または *tc* に指定されたコマンド群）を削除します。

enableCmd

void enableCmd (int cmd);
void enableCmd (const TCommandSet& tc);

この関数を呼び出したコマンドセットに，*cmd*（または *tc* に指定されたコマンド群）を追加します。

has

Boolean has (int cmd);

この関数を呼び出したコマンドセットに *cmd* が含まれていれば *True* を返します。

isEmpty

Boolean isEmpty ( );

この関数を呼び出したコマンドセットが空なら *True* を返します。

operator +=

void operator += (int cmd);
void operator += (const TCommandSet& tc);

enableCmd と同義です。この関数を呼び出したコマンドセットに *cmd*（または *tc* に指定されたコマンド群）を追加します。

operator −=

void operator −= (int cmd);

void operator −= (const TCommandSet& tc);

disableCmd と同義です。この関数を呼び出したコマンドセットから *cmd* (または *tc* に指定されたコマンド群) を削除します。

operator &= TCommandSet& operator &= (const TCommandSet& tc);

この関数を呼び出したコマンドセットを, *tc* との共通部分だけにして, その共通部分を返します。

operator |= TCommandSet& operator |= (const TCommandSet& tc);

そのコマンドセットと *tc* とで構成されるコマンドセット (集合で言う「和」) を返し, この関数を呼び出したコマンドセットをその「和」にします。

フレンド

operator & friend TCommandSet& operator & (const TCommandSet& tc1, const TCommandSet& tc1);

tc1 と *tc2* の共通部分 (両方のコマンドセットに共通なコマンド群) を返します。

operator == friend int operator == (const TCommandSet& tc1, const TCommandSet& tc2);

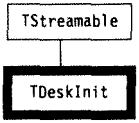
コマンドセット *tc1* と *tc2* が等しくない場合に *True* を返します。

operator != friend int operator != (const TCommandSet& tc1, const TCommandSet& tc2);

コマンドセット *tc1* と *tc2* が等しくない場合に *True* を返します。

operator | friend TCommandSet& operator | (const TCommandSet& tc1, const TCommandSet& tc2);

tc1 と *tc2* の「和」(両方の集合のいずれかに属しているコマンド群) を返します。



TDeskInit は，数多くのクラスの仮想基本クラスとして使用され，バックグラウンドオブジェクトを作成したり，挿入するときに，そのコンストラクタとメンバ関数 ***createBackground*** が利用されます。***TDeskInit*** の基本クラスは ***TStreamable*** です。したがって，フレンド関数や演算子を利用しながら，デスクトップをストリームに書き込んだり，ストリームから読み出したりすることができます。

メンバ関数

コンストラクタ　TDeskInit (TBackground * (*cBackground) (TRect bounds));

このコンストラクタは，引数に関数アドレスをとります。通常は，***& TDeskTop::initBackground*** です。***TDeskTop*** コンストラクタは，***TGroup***(*bounds*) と ***TDeskInit*** (***&TDesk::initBackground***) を呼び出して，サイズが *bounds* のデスクトップオブジェクトとそれにリンクさせたバックグラウンドを作成します。バックグラウンドは，デスクトップ・グループオブジェクトに挿入されます。

関連項目　***TDeskTop::TDeskTop***，***TDeskTop::initBackground***

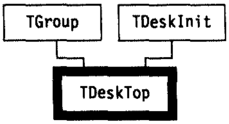
createBackground　TBackground * (*createBackground) (TRect bounds) ;

protected

TDeskInit コンストラクタから呼び出され，そのデスクトップで使用される，サイズが *bounds* のバックグラウンドオブジェクトを作成します。

関連項目 ***TDeskTop::TDeskTop***, ***TDeskTopInit::TDeskTopInit***

TDeskTop APP.H



TDeskTop は，***TGroup*** と仮想基本クラスである ***TDeskInit*** を多重継承します。バックグラウンドの作成および挿入には，***TDeskInit*** のコンストラクタと ***createBackground*** メンバ関数が使用されます。***TDeskTop*** は，アプリケーションウィンドウなどの他のビューのバックグラウンドとなる ***TBackground*** ビューを所有する単純なグループです。***TDeskTop*** は，画面の最上行にあるメニューバーから最下行にあるステータスラインまでの領域を占めます（ただしメニューバーとステータスラインが存在する場合のみ）。***TDeskTop*** オブジェクトは，オーバーロードされた >> 演算子と << 演算子を使用して，ストリームとの間で読み書きを行なうことができます。

データメンバ

background TBackground * background; **protected**

このデスクトップにリンクされている ***TBackground*** オブジェクトへのポインタ。

関連項目　***TDeskTop::initBackground***

メンバ関数

コンストラクタ　TDeskTop (const TRect& bounds) ;

基本のコンストラクタ ***TGroup*** (*bounds*) と ***TDeskInit(& TDeskTop::initBackground)*** を呼び出して，サイズが *bounds* の ***TDeskTop*** グループを作成します。そして，その際に作成された ***TBackgound*** オブジェクトをこのデスクトップに挿入します。***growMode*** は，***gfGrowHiX*** | ***gfGrowHiY*** に設定されます。

コンストラクタ　TDeskTop (StreamableInit streamableInit) ;　protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TDeskTop::initBackground***，***TGroup::TGroup***

build　static TStreamable * build () ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

cascade　void cascade (TRect& R) ;

そのデスクトップが所有しているすべてのタイル表示可能なウィンドウを，カスケード形式にして再表示します。Z オーダーに関して一番最初のタイル表示可能なウィンドウ（「奥にもっていかれる」ウィンドウ）は，デスクトップいっぱいまで拡大され，それ以降のウィンドウは，ひとつ奥のウィンドウよりも下方向に 1 行分，右方向

に 1 文字分ずつずらされます。有効なウィンドウは，一番小さなウィンドウとなって「最前面に」表示されます。

関連項目　***ofTileable***，***TDeskTop::tile***

handleEvent　virtual void handleEvent (TEvent& event) ;

TGroup::handleEvent を呼び出し，***cmNext*** コマンド(通常はホットキー ***F6***)と ***cmPrevious*** コマンドを検出した場合，そのデスクトップ下にある各ウィンドウを巡回させます（最初は，現在選択されているビューに渡される)。

関連項目　***TGroup::handleEvent***，***cmXXXX*** コマンド定数

initBackground　static TBackGround *initBackground (TRect) ;

このメンバ関数のアドレスが，***TDeskInit*** コンストラクタの引数として渡されます。***TDeskInit*** コンストラクタは，***initBackground*** を呼び出し，呼び出してきた ***TDeskTop*** オブジェクトと同じ *bounds* の ***TBackground*** オブジェクトを新たに作成します。***TDeskTop::background*** データメンバには，新しい ***TBackground*** オブジェクトへのポインタが書き込まれます。

関連項目　***TDeskTop::TDeskTop***

shutDown　virtual void shutDown ( ) ;

TObject::destroy が，導出された関連するオブジェクトを正しく消去するために，内部的に使用します。多くのクラスでは，***destroy*** が呼び出されたときに，関連するデータメンバを正しく処理できるように，***shutDown*** をオーバーライドしています。

関連項目　第 6 章**『信頼性の高いプログラムを開発するには』**

tile　void tile (TRect& r) ;

このデスクトップ下にあるすべての ***ofTileable*** ビューを，タイル形式で再表示します。

関連項目　***TDeskTop::cascade***，***ofTileable***

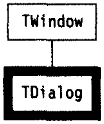
tileError　virtual void tileError ( ) ;

tileError は，***TDeskTop::tile*** または ***TDeskTop::cascade*** を呼び出して，エラーが返ってきたときに呼び出されます。デフォルトでは何もしません。これをオーバーライドして，アプリケーションでウィンドウの再配置ができないことをユーザに知らせるようにするのもよいでしょう。

関連項目　***TDeskTop::tile***，***TDeskTop::cascade***

関連する関数

演算子関数で ***TDisktop*** に関係するものもありますが，メンバ関数とはなっていません。詳細については322ページを参照してください。



TDialog は，***TWindow*** の直接の子であり，以下の性質を備えています。

- ***growMode*** は 0 とされます。すなわち，ダイアログボックスは拡大・縮小することはできません。
- データメンバ ***flags*** には，***wfMove*** と ***wfClose*** が設定されます。すなわちダイアログボックスは，移動することとクローズすること（クローズアイコンが付く）ができます。
- ***TDialog*** イベントハンドラは，***TWindow::handleEvent*** を呼び出しますが，それ以外に **ESC** キーや**リターン**キーの入力に対する特別な処理も行ないます。**ESC** キーが入力された場合には ***cmCancel*** コマンドを発行し，**リターン**キーが入力された場合には ***cmDefault*** コマンドを発行します。
- メンバ関数 ***TDialog::valid*** は，***cmCancel*** に対しては ***True*** を返します。***cmCancel*** でなかった場合には，そのグループの ***TGroup::valid*** を呼び出します。

メンバ関数

コンストラクタ　TDialog (const TRect& bounds, const char * aTitle);

以下の呼び出しを行なって，指定されたサイズおよびタイトルのダイアログボックスを作成します。

```
TWindow::TWindow(bounds, aTitle, wnNoNumber);
TWindowInit( &TDialog::initFrame);
```

growMode には 0 が，***flags*** には ***wfMove | wfClose*** が設定されます。したがってダイアログボックスは，デフォルトで，移動させること，およびクローズさせること（クローズアイコンによって）ができますが，拡大・縮小（サイズの変更）はできません。

コンストラクタ　TDialog (StreamableInit streamableInit) ;　protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

TDialog では特にデストラクタを定義していませんが，***TWindow***, ***TGroup***, および ***TView*** から継承した ***close*** とデストラクタを使用します。

関連項目　***TWindow::TWindow***

build　static TStreamable ＊ build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***, ***ipstream::readData***

getPalette　virtual TPalette& getPalette ( )　const ;

デフォルトのパレット文字列 ***cpDialog***, すなわち

"¥x20¥x21¥x22¥x23¥x24¥x25¥x26¥x27¥x28¥x29¥x2A¥x2B
¥x2C¥x2D¥x2E¥x2F¥x30¥x31¥x32¥x33¥x34¥x35¥x36¥x37
¥x38¥x39¥x3A¥x3B¥x3C¥x3D¥x3E¥x3F"

を返します。

handleEvent　virtual void handleEvent (TEvent& event) ;

TWindow::handleEvent (*event*) を呼び出し，さらに**リターン**

キーと **ESC** キーのイベントの処理を行ないます。**ESC** キーに対しては ***cmCancel*** コマンドが発行され，**リターン**キーに対しては ***cmDefault*** コマンドがブロードキャストされます。また，このメンバ関数では ***cmOK***，***cmCancel***，***cmYes***，および ***cmNo*** の各コマンドイベントの処理も行ない，ダイアログボックスのモードつき状態を終了させます。これらのイベントを正常に処理できた場合には，***clearEvent*** を呼び出します。

関連項目　***TWindow::handleEvent***

valid　virtual Boolean valid (ushort command) ;

引数 *command* が ***cmCancel*** であれば，*True* を返します。***cmCancel*** は，***handleEvent*** が，**ESC** キーを検出したときに発行するコマンドです。引数 *command* が ***cmCancel*** でなかった場合には，***valid*** は ***TGroup::valid***(*command*) を呼び出し，その呼び出しの結果を返します。***TGroup*** の ***valid*** は，そのグループ下の各サブビューの ***valid*** メンバ関数を呼び出します。したがって最終的に，***valid*** は，そのグループ下のすべてのコントロールが *True* を返してきた場合のみ *True* を返します。それ以外の場合には *False* を返します。***valid*** に対してすべてのサブビューが *True* を返すまでは，モードつき状態は終了しません。

関連項目　***TGroup::valid***

関連する関数

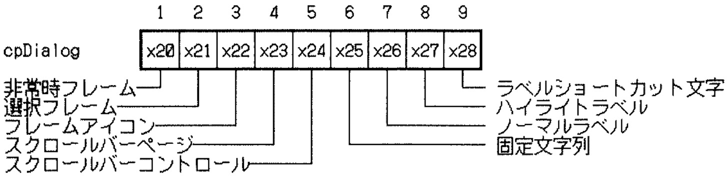
演算子関数で ***TDialog*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

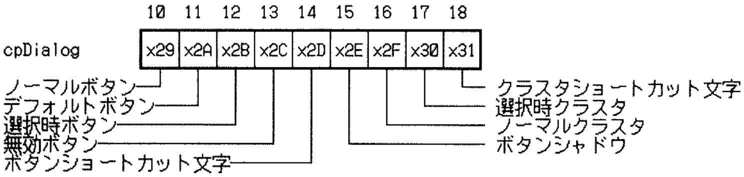
パレット

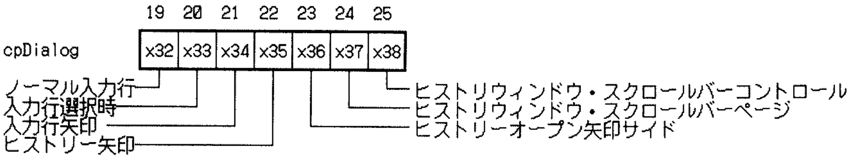
ダイアログボックスオブジェクトはデフォルトパレット ***cpDialog*** を使用し，アプリケーションパレットのエントリ32～36 に割り付け

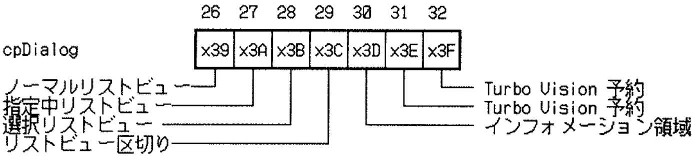
ます。

各クラスの getPalette メンバ関数も参照してください。

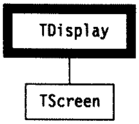








TDisplay SYSTEM.H



TDisplay は，その導出クラスである *TScreen* 用に低レベルの

ビデオ関数を用意します。この2つのクラスを含め SYSTEM.H にあるシステム関係のクラスについては，簡単にしか紹介しません。これらのクラスは，Turbo Vision が内部的に使用するものであり，通常のアプリケーションであれば直接使用する必要はありません。***TView*** は，***TDisplay*** のフレンドクラスです。

コンストラクタ TDisplay ( ) ; protected

TDisplay (const TDisplay&) ; protected

TDisplay オブジェクトを作成します。***TApplication*** コンストラクタを経由して自動的に呼び出されます。

デストラクタ ~TDisplay ( ) ; protected

TDisplay オブジェクトを消去します。

clearScreen static void clearScreen (uchar w, uchar h) ;

幅 *w*，高さ *h* の画面をクリアします。

getCols static ushort getCols ( ) ;

画面の桁数を返します。

getCrtMode static ushort getCrtMode ( ) ;

現在のビデオモードを返します。

関連項目 ***TDisplay::setCrtMode***，***TDisplay::videoModes***

getCursorType static ushort getCursorType ( ) ;

カーソルのタイプを返します。

関連項目 ***TDisplay::setCursorType***

getRows static ushort getRows ( ) ;

画面の行数を返します。

setCrtMode static void setCrtMode (ushort vmode) ;

(可能であれば)ビデオモードを *vmode* に設定します。使用できるハードウェアを調べ，ビデオモードを設定可能なモードのうち「最

善のモード」にします。

関連項目　***TDisplay::getCrtMode***, ***TDisplay::videoModes***

setCursorType　static void setCursorType (ushort ct) ;

カーソルのタイプを *ct* に設定します。

関連項目　***TDisplay::getCursorType***

videoModes

```
enum videoModes {
    smNColor   = 0x0001,
    smMono     = 0x0002,
    smEColor   = 0x0003
};
```

TDisplay が使用するビデオモードのニーモニック。

関連項目　***TDisplay::setCrtMode***, ***TDisplay::getCrtMode***

TDrawBuffer　　　　DRAWBUF.H

TDrawBuffer

TDrawBuffer はビューとならないクラスであり，表示バッファとの間で文字，属性，および文字列のやりとりを行なうバッファクラスです。表示バッファの内容は，通常 ***TView::writeBuf*** または ***TView::writeLine*** を呼んでテキストを表示します。

データメンバ

data　ulong data [maxViewWidth] ;　　protected

この表示バッファの配列を定義します。

メンバ関数

moveBuf

void moveBuf (ushort indent, const void far * source,
ushort attr, ushort count) ;

表示バッファにテキストを転送します。表示コード（ワード）と表示アトリービュート（ワード）のペアの配列をさす *source* から *count* 桁数分の表示コードのみがコピーされます。コピーは，*indent* で指定されたオフセットから開始されます。転送するときに *attr* を属性として設定しますが，0 の場合，属性部分は更新されません。

moveChar

void moveChar(ushort indent, ushort c, ushort attr, ushort count);

文字 *c* および属性 *attr* を *count* 回分，表示バッファの *indent* のオフセット以降に転送します。 *attr* が 0 でない場合には，転送するときに *attr* を属性として設定しますが，0 の場合，属性部分は更新されません。

関連項目 ***TDrawBuffer::putChar***

moveCStr

void moveCStr (ushort indent, const char far *str,
ushort attrs) ;

チルダ文字（'~'）を含む文字列 *str* を 2 色で表示バッファに転送します。コピーは， *indent* で指定されたオフセットから開始されます。*str* 中の各文字は，バッファの各ワードの下位バイトに設定されます。属性として，最初は *attr* の下位バイトを使用しますが，文字列中にチルダ文字があると属性の上位バイトを下位バイトを切り替えます。チルダ文字はコピーされません。

関連項目 ***TDrawBuffer::moveStr***

moveStr

void moveStr(ushort indent, const char far *str, ushort attrs);

文字列 *str* を呼び出し側の表示バッファに転送します。コピーは，*indent* で指定されたオフセットから開始されます。

attrs が 0 でない場合には転送するときに *attr* を属性として設定しますが，0 の場合には，属性部分は更新されません。

関連項目 ***TDrawBuffer::moveCStr***, ***TDrawBuffer::moveSubStr***

moveSubStr void MoveSubStr (ushort indent, ushort width,
const char far *str, ushort attr, ushort strofs) ;

文字列 *str* の *strofs* 桁目から *width* 桁数分の部分文字列を表示バッファに転送します。コピーは *indent* で指定されたオフセットから開始され，必ず *width* 桁数分転送されます。文字列 *str* の桁数が *strofs* 桁に満たないか，*strofs* 目から *width* 桁数分ないときには，空白がファイルされます。*attr* が 0 でない場合には，転送するときに *attr* を属性として設定します。

putAttribute void putAttribute (ushort indent, ushort attr) ;

バッファの *indent* で指定されたオフセットの位置に属性 *attr* を書き込みます。

関連項目 ***TDrawBuffer::putChar***

putChar void putChar (ushort indent, ushort c) ;

バッファの *indent* で指定されたオフセットの位置に文字 *c* を書き込みます。

関連項目 ***TDrawBuffer::moveChar***

フレンド

TSystemError と ***TView*** の 2 つのクラス，および関数 ***genRefs*** は，***TDrawBuf*** のフレンドです。

TEvent

構造体 ***TEvent*** の定義は以下のとおりです。

```
struct TEvent
{
   ushort what;
   union
   {
      MouseEventType mouse;
      KeyDownEvent keyDown;
      MessageEvent message;
   };
   void getMouseEvent();
   void getKeyEvent();
};
```

TEvent は ***what*** フィールドをキーにして，***MouseEventType*** 型，***KeyDownEvent*** 型，および ***MessageEvent*** 型の共用体をとります。***TView*** およびその導出クラスの ***handleEvent*** メンバ関数は，***TEvent*** 型の引数をとり，それに応じてしかるべき処理を行ないます。***TEvent*** のメンバ関数である ***getMouseEvent*** は，***TEventQueue::getMouseEvent*** を呼び出しイベントキューから適当なフィールドを取り出してきます。***TEvent::getKeyEvent*** は，キーイベントを DOS や BIOS を呼んで直接獲得します。

関連項目 ***MouseEventType***, ***KeyDownEvent***, ***MessageEvent***, ***TView::handleEvent***, ***TEventQueue***

TEventQueue

TEventQueue はマウスイベントの FIFO キューです。通常のアプリケーションでは，その内部的な詳細に関知する必要はまずありません。一般的には，*TEvent* 構造体と *TView::handleEvent* およびその下位クラスとの関係について知っていれば充分でしょう。

メンバ関数

コンストラクタ TEventQueue ( ) ;

TEventQueue オブジェクトを作成し，*resume* を呼び出します。

関連項目 *TEventQueue::resume*

デストラクタ ~TEventQueue ( ) ;

TEventQueue オブジェクトを消去し，*suspend* を呼び出します。

関連項目 *TEventQueue::suspend*

getMouseEvent static void getMouseEvent (TEvent& e) ;

FIFO キューにマウスイベントがあれば，1 個取り出し，そのデータメンバ *TEvent* にセットします。

関連項目 *TEvent::getMouseEvent*

resume static void resume ( ) ;

マウスが使えない場合には何もしません。マウスが使える場合には，*TMouse::getEvent* を呼び出して次のマウスイベントがあれば *TMouse::getEvent* でそれを獲得し，マウスハンドラを登録(また

は再登録）します。

関連項目　***TMouse::present***, ***TMouse::resvme***, ***TMouse::registerHandler***

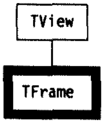
suspend　static void suspend（）；

現在の ***TMouse*** オブジェクトに対して，***TMouse::suspend*** を呼び出します。

関連項目　***TMouse::suspend***

フレンド

TView クラスおよび ***genRefs*** 関数は，***TEventQueue*** のフレンドです。



TFrame は，ウィンドウやダイアログボックスの周りに付けられるフレームです。フレームオブジェクトは，デフォルトでウィンドウオブジェクトに付加されるため，おそらくユーザがフレームオブジェクトを直接取り扱うことはないでしょう。

メンバ関数

コンストラクタ　TFrame (const TRect& bounds) ;

まず ***TView*** (*bounds*) を呼び出し，次に ***growMode*** を ***gfGrowHiX*** ｜ ***gfGrowHiY*** に設定し，さらに ***eventMask*** に ***eventMask*** ｜ ***evBroadcast*** を設定することによって，***TFrame*** オブジェクトがデフォルトでブロードキャストイベントを処理するようにします。

コンストラクタ　TFrame (StreamableInit streamableInit) ;　　**protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TView::TView***

build　static TStreamable *build () ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***，***ipstream::readData***，***TStreamable***

draw virtual void draw ( ) ;

現在の ***state*** フラグ（アクティブ，休止状態，ドラッグ中）に合わせた色属性およびアイコンを付けたフレームを表示します。オーナーウィンドウの ***flags*** にしたがって，ズーム，クローズ，リサイズの各アイコンを付加します。オーナーウィンドウにタイトルがあれば，そのデータメンバ ***title*** を読み出しタイトルを付加します。アクティブウィンドウは，太線のフレームで表示されアイコンが付加されます。休止状態のウィンドウは，細線のフレームで表示されアイコンは付加されません。

関連項目 ***sfXXXX*** 状態フラグ定数，***wfXXXX*** ウィンドウフラグ定数

getPalette virtual TPalette& getPalette ( ) const ;

デフォルトのフレームパレット文字列である ***cpFrame***，すなわち "¥x01¥x01¥x02¥x02¥x03" を返します。

handleEvent virtual void handleEvent (TEvent& event) ;

まず ***TView::handleEvent*** を呼び出し，その後マウスイベントの処理を行ないます。マウスがクローズアイコン上でクリックされた場合には，***TFrame::handleEvent*** は，***cmClose*** イベントを発行します。ズームアイコンをクリックするか，フレームの上部をダブルクリックした場合には ***cmZoom*** イベントが発行されます。またフレームの上部をドラッグすればウィンドウを移動させることができ，リサイズアイコンをドラッグすれば，そのビューの右下隅を動かすことによりそのビューのサイズを変えることができます。

関連項目 ***TView::handleEvent***

setState virtual void setState (ushort aState, Boolean enable) ;

TView::setState (*aState, enable*) を呼び出します。新しい状態が ***sfActive*** または ***sfDragging*** であれば，***drawView*** を呼び出してビューの再表示を行ないます。

関連項目　***TView::setState***，***TView::drawView***

フレンド

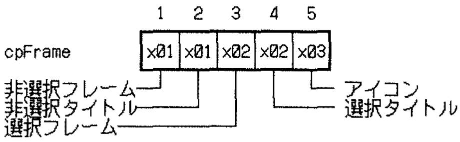
TDisplay は，***TFrame*** およびその導出クラスのフレンドです。

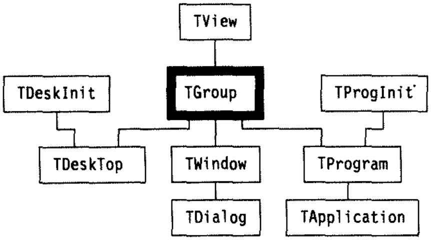
関連する関数

演算子関数で ***TFrame*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

フレームオブジェクトはデフォルトパレット ***cpFrame*** を使用し，標準ウィンドウパレットのエントリ 1〜3 に割り付けます。





TGroup オブジェクトおよびその導出クラス(略してグループといいます)は，Turbo Vision で中心的な役割を果たします。グループは特別な種類のビューです。***TView*** および ***TStreamable*** からすべてのメンバを継承するとともに，新たにメンバを設けたり多くのメンバでオーバーライドを行なうことによって，ビューのリンクリストを１個のオブジェクトのようにして動的に制御することができるようになっています。グループのサブビューが，それ自身グループである場合もよくあります。

グループは親の ***TView*** から矩形領域の境界を継承しますが，そのグループ下のサブビューを表示することによってのみそのグループを目にすることができます。概念的には，グループが自分自身を表示する場合，そのグループ下のサブビューのメンバ関数である ***draw*** を介して行ないます。グループはサブビューを所有しますが，それ以外にグループ全体の矩形領域の境界を表示（フィル）する機能も備えなければなりません。

サブビューやサブグループは，アプリケーションを起動してから終了するまでの間に生成され，グループに挿入され，ユーザの操作やアプリケーション自体が生成するイベントに基づいて表示されることになります。サブビューは，ユーザの操作（ウィンドウのクローズや，ダイアログボックスを抜けるなどの操作）によって，画面に表示しないようにしたり，グループから削除したり，廃棄したりが簡単に行なえます。

TGroup の３つのサブクラスである ***TWindow***, ***TDeskTop*** および ***TApplication***（***TProgram*** を経由したサブクラス）が，グ

ループやサブグループの概念を説明してくれます。***TApplication*** は，通常 ***TDeskTop***，***TStatusLine*** および ***TMenuView*** の各オブジェクトを所有します。***TDeskTop*** は ***TGroup*** のサブクラスなので，それ自体が ***TWindow*** オブジェクトを所有することができ，また ***TWindow*** も ***TFrame*** や ***TScrollBar*** などのオブジェクトを所有します。

第4章『**ビュー**』および第5章『**イベント駆動型プログラミング**』で説明してあるように，***TGroup*** の各オブジェクトは表示とイベント処理を配下のサブビューに任せます。

通常，***TGroup*** オブジェクトをインスタンス化する必要はありません。そうするかわりに，***TGroup*** のサブクラスである ***TApplication***，***TDeskTop***，および ***TWindow*** を1個または複数個インスタンス化します。

TGroup オブジェクトは，***TView*** を経由して ***TStreamable*** を継承しているので，すべてストリーム入出力可能なオブジェクトとなっています。したがって，***TGroup*** オブジェクト(アプリケーショングループ全体を含めて)は，C++ で通常使用する **iostream** 演算子を使用して，型に正しく対応させながらストリームとの間で読み書きを行なうことができます。

データメンバ

buffer　void far *buffer;

キャッシュ再表示を行なう際に使用されるバッファを指します。グループにキャッシュバッファがない場合には 0 です。グループの ***opotions*** メンバの ***ofBuffered*** フラグがクリアされていなければ，キャッシュバッファの生成と消去は自動的に行なわれます。

関連項目　***TGroup::draw***，***TGroup::lock***，***TGroup::unlock***

clip　TRect clip;

グループのクリップ領域(***getExtent*** または ***getClipRect*** が返してくる)を格納します。クリップ領域とは，***draw*** が呼び出された

ときに再表示する必要のある最小範囲のことです。

関連項目　***TView::getClipRect***，***TView::getExtent***

current　TView　*current;

現在選択されているサブビューを指します。サブビューがどれも選択されていない場合には 0 です。

関連項目　***sfSelected***，***TView::select***

endState　ushort endState;

endModal の呼び出しを行なった後のグループの状態が入ります。

関連項目　***TGroup::endModal***

last　TView　*last;

グループの最後のサブビュー（Z オーダーでみて手前側からもっとも遠いサブビュー）を指します。

lockFlag　uchar lockFlag;

バッファを使用したグループ表示を制御する際のセマフォの役割を果たします。***lockFlag*** には，***draw*** 呼び出しがネストする場合に設定されるロックの回数が格納されます。***lock*** および ***unlock*** がこの値を増減させます。この値がゼロになると，グループは，グループ全体をバッファデータを使用して表示します。表示を繰り返し行なうような場合には，***lock*** 呼び出しと ***unlock*** 呼び出しでその処理ブロックを挟むことによって，画面の「どたばた」が余り起こらないようにしたほうがよいでしょう。

関連項目　***TGroup::draw***，***TGroup::lock***，***TGroup::unlock***

phase　phaseType phase;

フォーカスイベント処理の現在のフェーズ。***ofPreProcess*** フラグまたは ***ofPostProcess*** フラグを立てているサブビューは，***owner->phase*** を調べることによって，サブビューの ***handleEvent*** に対する呼び出しが ***phPreProcess***，***phFocused***，***phPostProcess*** のフェーズのうちのどれであるかを確かめること

ができます。

phaseType は，***TView*** で列挙型として，次のように定義されています。

```
enum phaseType {phFocussed, phPreProcess, phPostProcess};
```

関連項目　***ofPreProcess***, ***ofPostProcess***, ***TGroup::handleEvent***

メンバ関数

コンストラクタ　TGroup (TRect& bounds) ;

TView::TView (*bounds*) を呼び出し，***options*** の ***ofSelectable*** と ***ofBuffered*** を立て，***eventMask*** に 0xFFFF を設定します。***last***, ***current***, ***buffer***, ***lockFlag***, および ***endState*** の各メンバは，全部 0 に設定されます。

コンストラクタ　TGroup (StreamableInit streamableInit) ;　　protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TView::TView***

デストラクタ　~TGroup ( ) ;

hide を使用してグループを画面から消し，***delete p*** を使用してグループ内の各サブビューを廃棄していきます。最後に，バッファがあればそのバッファを解放します。

関連項目　***TView::hide***

at TView *at (short index) ;

Z オーダーでみて *index* の位置にあるサブビューへのポインタを返します。

関連項目 ***TGroup:indexOf***

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***, ***ipstream::readData***

changeBounds virtual void changeBounds (TRect& bounds) ;

TView::changeBounds をオーバーライドします。グループの境界を *bounds* に変更した後，***calcBounds*** を呼び出し，さらにグループ内の各ビューの ***changeBounds*** を呼び出します。

関連項目 ***TView::calcBounds***, ***TView::changeBounds***

dataSize virtual ushort dataSize ( ) ;

TView::dataSize をオーバーライドします。各サブビューの ***dataSize*** を呼び出してサイズを積算していき，グループ全体のサイズを返します。

関連項目 ***TView::dataSize***

draw virtual void draw ( ) ;

TView::draw をオーバーライドします。キャッシュバッファがある場合には（データメンバ ***TGroup::buffer*** を参照），***TView::writeBuf*** を使用してバッファデータを画面に書き込みます。キャッシュバッファがない場合には，***TGroup::redraw*** 呼び出しを行なってサブビューごとに自分自身の表示を行なわせます。

関連項目 ***TGroup::buffer***, ***TGroup::redraw***, ***TView::draw***

drawSubViews void drawSubViews (TView *p, TView *bottom) ;

TG
Std class

p* から **bottom* までの各サブビューの ***drawView* を呼び出します。

endModal virtual void endModal (ushort command);

endModal は，このグループが現在のモードつきビューであればモードつき状態を終了させます。*command* が ***execView*** に渡されます (***execView*** によってこのビューはモードつきにされた)。***execView*** は，戻り値として *command* を返してきます。このグループが現在のモードつきビューで**ない**場合には ***TView::endModal*** を呼び出します。

関連項目 ***TGroup::execView***，***TGroup::execute***，***TView::endModal***，***sfModal***

eventError virtual void eventError (TEvent& event);

eventError は，モードつきの ***TGroup::execute*** 内のイベント処理ループにおいて処理することのできないイベントが検出されるたびに呼び出されます。デフォルトの処理内容は次のとおりです。そのグループのオーナーがゼロでない場合に ***eventError*** は，そのオーナーの ***eventError*** を呼び出します。通常この呼び出しは，***TApplication*** の ***eventError*** にまで遡ってチェインしていきます。***eventError*** をオーバーライドして，何か適当な処理を行なわせるようにすることもできます。

関連項目 ***TGroup::execute***，***TGroup::execView***，***sfModal***

execute virtual ushort execute ();

TView::execute をオーバーライドします。***execute*** は，グループのメインのイベントループです。***getEvent*** を呼び出してイベントを獲得し，***handleEvent*** を呼び出してそのイベントを処理するということを繰り返します。このイベントループは，そのグループまたは別のサブビューが ***endModal*** を呼び出すことによって終了します。ただし，処理を戻す前に ***valid*** を呼び出し本当にモードつき状態を終了してよいかどうかを確認します。

関連項目 ***TGroup::getEvent***，***TGroup::handleEvent***，

TGroup::endModal, ***TGroup::valid***, ***TGroup::endState***

execView ushort execView (TView *p) ;

execView は,「モードなしの」メンバ関数 ***insert*** および ***remove*** の「モードつき」版です。***insert*** の場合とは異なり, グループにビューを挿入した後, ***execView*** はそのビューの実行が終了するのを待ち, 終了したらビューを削除し, 最後にその実行結果を返します。***execView*** は, Turbo Vision のあちこちで使用されます。その顕著な例が ***TApplication::run*** およびモードつきダイアログボックスの実行における使用です。

TG Std class

execView は, まず現在のコンテキスト (選択されているビュー, モードつきビュー, およびコマンドセット)をセーブし, 次に *p*−> ***setState*** (***sfModal***, ***True***) を呼び出して *p* をモードつきにし, そのグループに *p* を挿入します (まだ挿入されていない場合)。その後 *p*−>***execute*** を呼び出します。*p*−>***execute*** が処理を戻してきたら, そのグループを以前の状態に復旧し, *p*−>***execute*** の結果が ***execView*** 呼び出しの結果として戻されます。*p* を 0 にして ***execView*** を呼び出した場合には, ***cmCancel*** が戻り値となります。

関連項目 ***TGroup::insert***, ***TGroup::execute***, ***sfModal***

first TView *first () ;

グループの最初のサブビュー (Z オーダーでみて手前側にもっとも近いサブビュー) へのポインタを返します。グループにサブビューがない場合には, 0 を返します。

関連項目 ***TGroup::last***

firstMatch TView *firstMatch (ushort aState, ushort aOptions)

state が ***aState*** と一致し, ***options*** が ***aOptions*** と一致する最初のサブビューへのポインタを返します。

firstThat TView *firstThat (Boolean (*func) (TView* , void*), void *args) ;

firstThat は，ユーザ側で用意した ***Boolean*** 型の関数 **func* を，*args* で引数リストを指定し(指定しなくてもよい)，グループ内の各サブビューに適用していきます。**func* が *True* を返してくるまで続けられます。戻り値は，**func* が *True* を返したサブビューへのポインタです。すべてのサブビューで *False* が返された場合には，戻り値は 0 です。

func* の第 1 引数には，サブビューへのポインタが入れられていきます。func* の第 2 引数には，***firstThat*** の引数であるポインタ *args* が設定されます。***firstThat*** は以下のようなつくりになっています。

```
TView *TGroup::firstThat( Boolean (*func)(TView *, void *), void *args )
{
    TView *temp = last;
    if( temp == 0 )
        return 0;

    do  {
        temp = temp->next;
        if( func( temp, args ) == True )
            return temp;
    } while( temp != last );
    return 0;
}
```

関連項目　***TGroup::forEach***

forEach　void　forEach (void　(*func) (TView　*),
void　*), void　*args) ;

forEach は，関数 **func* で指定された処理をグループ内の各サブビューに対して Z オーダーで適用していきます。以下に示されるように，引数 *args* によって処理関数に任意の引数を渡すことができます。

```
void TGroup::forEach( void (*func)(TView*, void *), void *args )
{
   TView *term = last;
   TView *temp = last;
   if( temp == 0 )
      return;

   TView *next = temp->next;
   do  {
      temp = next;
```

```
        next = temp->next;
        func( temp, args );
    } while( temp != term );

}
```

関連項目 ***TGroup::firstThat***

freeBuffer

void freeBuffer ( ) ;

グループに表示バッファがある場合に ***delete buffer*** を呼び出し，***buffer*** に 0 を設定することによってバッファを解放します。

関連項目 ***TGroup::buffer***，***TGroup::getBuffer***，***TGroup::draw***

getBuffer

void getBuffer ( ) ;

グループが ***sfExposed*** かつ ***ofBuffered*** である場合に，表示バッファを作成します。バッファのサイズは (*size.x* **size.y*) < とされ，データメンバ ***buffer*** には新しいバッファへのポインタが入ります。

関連項目 ***TGroup::buffer***，***TGroup::freeBuffer***，***TGroup::draw***

getData

virtual void getData (void *rec) ;

TView::getData をオーバーライドします。Z オーダーとは逆順に，すなわち *rec* で指定した位置を各サブビューの ***dataSize*** 分ずつインクリメントしながら，各サブビューの ***getData*** を呼び出していきます。

関連項目 ***TView::getData***，***TGroup::setData***

getHelpCtx

virtual ushort getHelpCtx ( ) ;

選択されているサブビューの ***getHelpCtx*** メンバ関数を呼び出すことによって，現在のフォーカスビューのヘルプコンテキストを返します。どのサブビューもヘルプコンテキストを指定していない場合には，***getHelpCtx*** は自分自身の ***HelpCtx*** メンバを返します。

handleEvent　virtual void handleEvent（TEvent& event）;

TView::handleEvent をオーバーライドします。基本的にグループのイベント処理は，その配下にある1個以上のサブビューにイベントを渡すことによって行なわれます。ただし，イベントを渡す実際の経路は，イベントクラスに左右されます。

フォーカスイベント（デフォルトでは，***evKeyDown*** および ***evCommand***。***focusedEvents*** 変数を参照）については，イベント処理は3つのフェーズに分かれます。第1フェーズではグループの ***phase*** メンバを ***phPreProcess*** にセットし，***ofPreProcess*** フラグの立っている全サブビューの ***handleEvent*** にそのイベントを渡します。次のフェーズでは ***phase*** を ***phFocused*** にセットし，現在選択されているビューの ***handleEvent*** にそのイベントを渡します。最後のフェーズでは ***phase*** を ***phPostProcess*** にセットし，***ofPostProcess*** フラグの立っている全サブビューの ***handleEvent*** にそのイベントを渡します。

ポジショナルイベント（デフォルトでは ***evMouse***。***PositionalEvents*** 変数を参照）については，イベントは *event.where* で指定される点を包含する領域を占めている最初のサブビューの ***handleEvent*** に渡されます。

ブロードキャストイベント（フォーカスイベントでもポジショナルイベントでもないイベント）については，グループ内の1つ1つのサブビューの ***handleEvent*** に，Z オーダーで渡されていきます。

サブビューの ***eventMask*** メンバが，あるイベントクラスをマスクして排除している場合，***TGroup::handleEvent*** は，そのクラスのイベントをそのサブビューには**決して送出しません**。たとえば ***TView*** のデフォルトの ***eventMask*** では，***evMouseUp*** と ***evMouseMove*** と ***evMouseAuto*** が無効にされているので，***TGroup::handleEvent*** が標準の ***TView*** にその種のイベントを送出することは決してありません。

関連項目　***focusedEvents***，***positionalEvents***，***evXXXX*** イベント定数，***TView::eventMask***，***handleEvent*** メンバ関数

indexOf short indexOf (TView *p) ;

サブビュー **p* の Z オーダーでの位置（インデックス）を返します。

関連項目 ***TGroup:at***

insert void insert (TView *p) ;

p で指定したビューをグループのサブビューリストに挿入します。新たに挿入されるサブビューは，他のすべてのサブビューよりも上に配置されます。サブビューの ***ofCenterX*** フラグまたは ***ofCenterY*** フラグが立っている場合にはそのフラグにしたがって中央に配置されます。***sfVisible*** フラグが立っている場合には，グループ内に表示され，そうでない場合には表示の指定があるまで表示されません。***ofSelectable*** フラグが立っている場合には，そのサブビューが現在選択されているサブビューとなります。

関連項目 ***TGroup::remove***, ***TGroup::execView***

insertBefore void insertBefore (TView *p, TView *target) ;

p で指定されるビューを *target* で指定されるビューの前に挿入します。*target* が 0 の場合には，グループ内の他のすべてのサブビューよりも後ろに配置されます。

関連項目 ***TGroup::insert***, ***TGroup::remove***

lock void lock ( ) ;

グループをロックし，ロックが解除されるまでサブビューによる画面への書き込みを保留します。グループがキャッシュバッファを使用していない場合には（***ofBuffered*** および ***TGroup::buffer*** を参照），***lock*** は何も行ないません。***lock*** は，データメンバ ***lockFlag*** をインクリメントすることによって機能します。これはセマフォになっており，同様にして ***unlock*** によってデクリメントされます。***unlock*** 呼び出しによってカウンタが 0 になると，キャッシュバッファに保存されていたイメージを使って，グループ全体が画面に書き込まれます。

TG
Std class

表示中心の処理を ***lock*** 呼び出しと ***unlock*** 呼び出しで「挟む」ことによって，見苦しい「画面のどたばた」を完全に除去することはできないにしろ，減少させることはできます。たとえば，メンバ関数 ***TDeskTop::tile*** や ***TDeskTop::cascade*** が ***lock***，***unlock*** を使用してどたばたを減少させています。

☞ ***lock*** 呼び出しと ***unlock*** 呼び出しの回数は一致させる**必要があります**。一致しない場合には，グループはずっとロックした状態となり，再表示が要求された場合でも正しく再表示が行なわれないことになります。

関連項目　***TGroup::unlock***，***TGroup::lockFlag***

matches　Boolean matches (TView *p)；

ビュー **p* の ***state*** および ***options*** の設定が，呼び出しを行なったビューのものと同じであれば *True* を返します。

read　virtual void *read (ipstream& is)；

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream***

redraw　void redraw ( )；

グループの配下にあるサブビューを Z オーダーで再表示します。***TGroup::redraw*** と ***TGroup::draw*** とは，***redraw*** がキャッシュバッファを使用することがないという点で異なります。

関連項目　***TGroup::draw***

remove　void remove (TView *p)；

グループからサブビュー *p* を削除し，必要なら残りのサブビューの再表示を行ないます。*p* のデータメンバ ***owner*** と ***next*** に 0 をセットします。

関連項目　***TGroup::insert***，***TGroup::removeView***

removeView　void removeView (TView *p)；

グループからサブビュー *p* を削除します。***TGroup::remove*** が内部的に使用します。

関連項目　***TGroup::remove***

resetCurrent　void resetCurrent ( ) ;

チェイン中のサブビューのうち，***sfVisible*** かつ ***ofSelectable*** である最初のサブビューを選択（カレントのサブビューに）します。***resetCurrent*** の内容は次のとおりです。

```
setCurrent(firstMatch(sfVisible, ofSelectable), normalSelect).
```

選択モード引数用として，以下の enum 型が定義されています。

```
enum selectMode { normalSelect, enterSelect, leaveSelect };
```

関連項目　***TGroup::setCurrent***

selectNext　void selectNext (Boolean forwards) ;

forwards が *True* の場合には，そのグループにおいて Z オーダーでみて次に選択可能なサブビュー（***ofSelectable*** ビットの立っているサブビュー）を選択(カレントのサブビューに)します。*forwards* が *False* の場合には，１つ前の選択可能なサブビューを選択します。

関連項目　***ofXXXX*** オプションフラグ定数，***TGroup::selectView***

setCurrent　void setCurrent (TView *p, selectMode mode) ;

selectMode は，***TGroup*** において列挙型として次のように定義されています。

```
enum selectMode {normalSelect, enterSelect, leaveSelect};
```

p* が現在のサブビューである場合には，*setCurrent*** は何も行ないません。**p* が現在のサブビューでない場合には，***setState*** を呼び出すことによって **p* を現在のサブビュー（すなわち選択されて

いるサブビュー）にします。

関連項目　***TGroup::resetCurrent***

setData　virtual void setData (void *rec)；

TView::setData をオーバーライドします。Z オーダーとは逆順に，すなわち *rec* で指定した位置を各サブビューの ***dataSize*** 分ずつインクリメントしながら各サブビューの ***setData*** を呼び出していきます。

関連項目　***TGroup::getData***，***TView::setData***

setState　virtual void setState (ushort aState, Boolean enable)；

TView::setState をオーバーライドします。継承した ***TView::setState*** を呼び出してから以下のようなサブビューの更新を行ないます。

- *aState* が ***sfActive*** か ***sfDragging*** である場合には，各サブビューの ***setState*** を呼び出し，*aState* の内容どおりにサブビューの更新を行ないます。
- *aState* が ***sfFocused*** である場合には，現在選択されているサブビューを呼び出し，*aState* の内容どおりに自分自身にフォーカスを当てさせます。*aState* が ***sfExposed*** である場合には，各サブビューの ***doExpose*** が呼び出されます。最後に，*enable* が *False* であれば ***freeBuffer*** を呼び出します。

関連項目　***TView::setState***，***TGroup::doExpose***，***TGroup::freeBuffer***

shutDown　virtual void shutDown ( )；

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するため内部的に使用します。***shutDown*** は，***destroy*** が呼び出されたときに，関連するデータメンバの設定を適切に行なえるようにするため多くのクラスでオーバーライドされます。

関連項目　第 6 章『**信頼性の高いプログラムを開発するには**』

unlock　void unlock ( ) ;

lockFlag をデクリメントしてグループのロックを解除します。その結果 lockFlag がゼロになった場合には，キャッシュバッファに保存されていたイメージを使ってグループ全体を画面に書き込みます。

関連項目　***TGroup::lock***

valid　virtual Boolean valid (ushort command) ;

TView::valid をオーバーライドします。すべてのサブビューに対する ***valid*** 呼び出しで，*True* が返ってきた場合には *True* を返します。***TGroup::valid*** は，***TGroup::execute*** のイベント処理ループの最後のところで終了してもよいことを確認するために使用されます。モードつき状態は，すべての ***valid*** 呼び出しが *True* を返してきたときでなければ終了することはできません。サブビューは，制御権を握っていたい場合には *False* を返すこともできます。

関連項目　***TView::valid***，***TGroup::execute***

write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

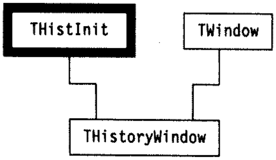
関連項目　***TStreamableClass***，***TStreamable***，***opstream***

フレンド

関数 ***genRefs*** は，***TGroup*** のフレンドです。

関連する関数

演算子関数で ***TGroup*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。



THistInit には，リストを作成するコンストラクタとそれをヒストリーウィンドウに挿入する ***createListViewer*** メンバ関数があります。

メンバ関数

コンストラクタ

THistInit(TListViewer * (*cListViewer)
(TRect r, TWindow *w,ushort histID)；

THistoryWindow コンストラクタが ***cListViewer*** 型である ***THistory::initViewer*** のアドレスを引数にして，基本のコンストラクタであるこの ***THistInit::THistInit*** を呼び出します。このコンストラクタは，指定したサイズおよびヒストリーリストを備えたリストビュアーを作成し，指定されたヒストリーウィンドウに挿入します。

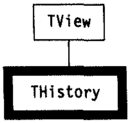
関連項目 ***THistoryWindow*** コンストラクタ

createListViewer

TListViewer *(*createListViewer) (TRect r, TWindow *w,
ushort histId)； **protected**

THistInit コンストラクタから呼び出され，*histId* で指定されたヒストリーリストで構成されるサイズ *r* のウィンドウ *w* 用のリストビュアーを作成します。

関連項目 ***THistory***，***TListViewer***，***THistoryWindow***



THistory オブジェクトは，以前行なったデータ入力，操作，選択からなる選択リストを作成し，ユーザがそれを「再度実行」することができるようにします。***THistory*** オブジェクトは，***TInputLine*** オブジェクトおよびヒストリーリストにリンクされます。ヒストリーリストの情報は，ヒープ上のメモリブロックに格納されます。ブロックが満杯になったときには，もっとも古いヒストリー項目が削除され新しい項目が追加されます。

THistory そのものは，入力行につづくアイコン（[↓]）で表示されます。ユーザがヒストリーアイコンをクリックすると，Turbo Vision は以前に入力されたデータのリストを内容とするヒストリービュアー（***THistoryViewer*** を参照）の入ったヒストリーウィンドウ（***THistoryWindow*** を参照）をオープンします。

異なる入力行であっても，同じ ID 番号を使用することにより，同じヒストリーリストを共有することができます。

データメンバ

historyID ushort historyId ;

各ヒストリーリストには，プログラムで指定した一義的な ID 番号が割り当てられます。別々のウィンドウに作られた別個のヒストリーオブジェクトであっても，同じ ID 番号を使用することによってヒストリーリストを共有することができます。

link TInputLine *link ;

リンクされた ***TInputLine*** オブジェクトへのポインタ。

メンバ関数

コンストラクタ　THistory(const TRect& bounds, TInputLine *aLink, ushort aHistoryId) ;

TView (*bounds*) を呼び出して指定されたサイズの ***THistory*** オブジェクトを作成し，指定された引数をデータメンバ ***link*** および ***historyId*** に設定します。メンバ ***options*** には ***ofPostProcess*** 設定されます。***eventMask*** には ***evBroadcast*** ビット，および ***evMouseDown***, ***evKeyDown*** が立てられます。また ***evCommand*** の各ビットも ***TView*** (*bounds*) によって立てられます。

コンストラクタ　THistory (StreamableInit streamableInit) ;　　protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに, vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TView::TView***

build　static TStreamable *build () ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***, ***ipstream::readData***

draw　virtual void draw () ;

デフォルトパレットを使用して ***THistory*** アイコンを表示します。

getPalette　virtual TPalette& getPalette () const ;

デフォルトパレットである ***cpHistory***，すなわち "¥x16¥x17" へのポインタを返します。

handleEvent　virtual void handleEvent (TEvent& event) ;

TView::handleEvent (*event*) を呼び出した後，関係のあるマウスイベントおよびキーイベントの処理を行なってリンクしている入力行を選択し，ヒストリーウィンドウを作成します。

関連項目　***THistory::initHistoryWindow***

initHistoryWindow　virtual THistoryWindow *initHistoryWindow(const TRect& bounds)

THistoryWindow オブジェクトを作成し，そのオブジェクトへのポインタを返します。新しいオブジェクトは指定された *bounds* を持ち，呼び出し側の ***THistory*** オブジェクトと同じ ***historyId*** が与えられます。また ***helpCtx*** は，呼び出し側のオブジェクトにリンクしている ***TInputLine*** のものが設定されます。

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream***

shutDown　virtual void shutDown ( ) ;

TObject::destroy が導出したオブジェクトや，関連するオブジェクトを間違いなく消去するために内部的に使用します。***shutDown*** は，***destroy*** が呼び出されたときに，関連するデータメンバの設定を適切に行なえるようにするために，多くのクラスでオーバーライドされます。

関連項目　第 6 章**『信頼性の高いプログラムを開発するには』**

write　virtual void write (opstream& os) ;

出力ストリーム os への書き込みを行ないます。

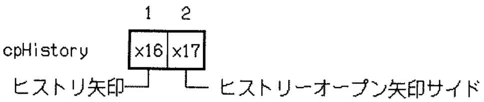
関連項目　***TStreamableClass***，***TStreamable***，***opstream***

関連する関数

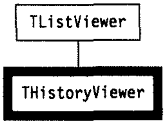
演算子関数で ***THistory*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

History アイコンはデフォルトパレット ***cpHistory*** を使用し，標準ダイアログボックスのパレットのエントリ 22～23 に割り付けます。



THistoryViewer DIALOGS.H



THistoryViewer は，***TListViewer*** 直系の下位クラスです。ヒストリーリストシステムで使用され，ヒストリーアイコンをクリックすると作成されるヒストリーウィンドウ内に表示されます。***THistory***，***THistoryWindow***，および ***THistoryViewer*** の連携機能については，407 ページの ***THistory*** の項で詳細に説明してあります。

データメンバ

historyId ushort historyId ;

historyId は，ビュー中に表示されるヒストリーリストの ID 番号です。

メンバ関数

コンストラクタ THistoryViewer (const TRect& bounds, TScrollBar *aHScrollBar, TScrollBar *aVScrollBar, ushort aHistoryId) ;

まず ***TListViewer*** コンストラクタを呼び出し，ビュアーリストを初期化します。すなわち，境界，段数 1， *aHScrollBar* および *aVScrollBar* で渡された 2 つのスクロールバーのポインタをセットします。次に ***historyId*** データメンバに *aHistoryId* で渡された値を設定し，そのビューをヒストリーリストにリンクします。次にリストの長さを調べ，リストの範囲をそのリストの項目数とします。ヒストリーリストの最初の項目にフォーカスを当て，水平スクロールの範囲をリスト中でもっとも幅の広い項目に合わせます。

関連項目 ***TListViewer::TListViewer***

getPalette virtual TPalette& getPalette ( ) const ;

デフォルトパレット文字列 ***cpHistoryViewer***，すなわち "¥x06¥x06¥x07¥x06¥x06" を返します。

getText virtual void getText (char *dest, short item, short maxLen) ;

リンクさせてあるヒストリーリスト中の *item* 番目の文字列を *dest* にセットします。 ***getText*** は，リスト内の表示可能な各項目の仮想メンバ関数 ***draw*** から呼び出されます。

関連項目 ***TListViewer::draw***

TH
Std class

handleEvent　virtual void handleEvent（TEvent& event）;

ヒストリービュアーが処理するイベントには，2種類のイベントがあります。その他のイベントは，***TListViewer::handleEvent*** に渡されます。ダブルクリックまたは**リターン**キーの入力があると ***cmOK*** コマンドを発行し，ヒストリーウィンドウのモードつき状態を終了させます。***ESC*** キーが入力された場合，または ***cmCancel*** コマンドイベントが発生した場合には，ヒストリーリストからの選択をキャンセルします。

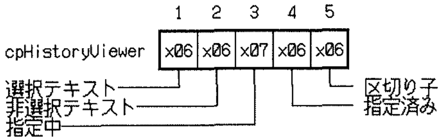
関連項目　***TListViewer::handleEvent***

historyWidth　int historyWidth（）;

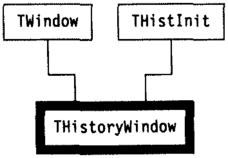
historyId でリンクされているヒストリーリスト中で最長の文字列の長さを返します。

パレット

ヒストリービュアーオブジェクトは，デフォルトパレット ***cpHistoryViewer*** を使用し，標準ダイアログボックスのパレットのエントリ 6～7 に割り付けます。



THistoryWindow　　dialogs.h



ヒストリーリストの使用法やヒストリーリストと関係するオブジェクトについては，407ページのTHistory の項で詳しく説明してあります。

THistoryWindow は，***TWindow*** と ***THistInit*** を特殊化した下位クラス（多重継承）であり，ユーザが入力行の隣にあるヒストリーアイコンをクリックした際，ヒストリーリストビュアーを入れるために使用されます。とくに指定しなければ，このウィンドウにはタイトルと番号は付きません。ヒストリーウィンドウのフレームには，クローズアイコンだけが付きます。したがってクローズすることはできますが，リサイズ（拡大・縮小）を行なうことはできません。

データメンバ

viewer　TListViewer　*viewer ;　**protected**

このヒストリーウィンドウに入るリストビュアーへのポインタ。

メンバ関数

コンストラクタ　THistoryWindow (const TRect& bounds, ushort aHistoryId) ;

&THistoryWindow::initViewer を引数にして ***THistInit*** コンストラクタを呼び出します。これによってリストビュアーが作成されます。次に ***TWindow*** コンストラクタを呼び出し，指定された

bounds を持ち，タイトル文字列がヌルでウィンドウ番号のないウィンドウ（***wnNoNumber***）を作成します。次に ***&THistoryWindow::initFrame*** を引数にして ***TWindowInit*** コンストラクタを呼び出し，ヒストリーウィンドウのフレームを作成します。最後に，***TWindow::flags*** データメンバに ***wfClose*** を設定してクローズアイコンを付け，ヒストリービュアーオブジェクトを作成してヒストリーウィンドウに挿入し，*aHistoryId* で指定されたヒストリーリストの項目を表示します。

関連項目　***TWindow*** コンストラクタ，***THistoryWindow::initViewer***

getPalette　virtual TPalette& getPalette ( )　const ;

デフォルトパレット文字列 ***cpHistoryWindow***，すなわち "¥x13¥x13¥x15¥x18¥x17¥x13¥x14¥" を返します。

getSelection　virtual void getSelection (char *dest) ;

リンクしているヒストリービュアーにおいて，フォーカスを得ている項目の文字列を *dest* に返します。

関連項目　***THistoryViewer::getText***

initViewer　static TListViewer *initViewer (TRect bounds, TWindow * w, ushort aHistoryId) ;

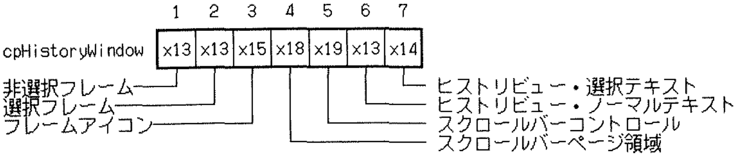
THistoryViewer オブジェクトをインスタンス化し，***aHistoryId*** という ID でリンクしているリスト用のヒストリーウィンドウ内に挿入します。ウィンドウのフレームに標準スクロールバーを配置し，リストをスクロールさせることができるようにします。

関連項目　***THistoryViewer*** コンストラクタ

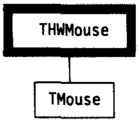
パレット

ヒストリーウィンドウオブジェクトはデフォルトパレット ***cpHistoryWindow*** を使用し，標準ダイアログボックスのパレットのエ

ントリ 19～25 に割り付けます。



THWMouse SYSTEM.H



THWMouse は，その導出クラスである ***TMouse*** 用に低レベルのマウス処理関数を用意します。この 2 つのクラスを含め SYSTEM.H にあるシステム関係のクラスについては，簡単にしか紹介しません。これらのクラスは，Turbo Vision が内部的に使用するものであり，通常のアプリケーションであれば直接使用する必要はありません。

データメンバ

buttonCount static uchar near buttonCount; **protected**

マウスのボタン数が入ります。マウスが検出されなかった場合は 0 です。

メンバ関数

コンストラクタ THWMouse ( ); **protected**

THWMouse (const THWMouse& m) ;　　protected

THWMouse::resume を呼び出します。

関連項目　***THWMouse::resume***

デストラクタ　~THWMouse ( ) ;　　protected

THWMouse::suspend を呼び出します。

関連項目　***THWMouse::suspend***

getEvent　static void getEvent (MouseEventType& me) ;　　protected

イベントキューからマウスイベントを取り出し, ***MouseEventType*** 構造体である *me* のデータメンバ ***buttons***, ***where.x***, ***where.y*** および ***doubleClick*** にデータを設定します。

関連項目　***MouseEventType***

hide　static void hide ( ) ;　　protected

マウスカーソルを表示しないようにします。

present　static Boolean present ( ) ;　　protected

マウスが物理的に存在し，有効になっていれば *True* を返します。そうでなければ *False* を返します。

registerHandler　static void registerHandler (unsigned mask, void (far *func) ()) ;　　protected

func を現在有効なマウスハンドラとして登録し，***handlerInstalled*** を *True* にします。

resume　static void resume ( ) ;　　protected

ハンドラを (再) 登録し，***buttonCount*** を (再) 設定することによってマウスを復旧します。

setRange　static void setRange (ushort rx, ushort ry) ;　　protected

マウスの移動可能範囲を左上 (0,0) から引数に指定された 右下 (*rx*，*ry*) に設定します。

show static void show ( ) ; protected

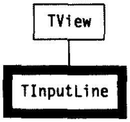
マウスカーソルを表示します。

suspend static void suspend ( ) ; protected

present が ***False*** を返した場合には何もしません。***True*** を返した場合にはマウスの表示をやめ，ハンドラを登録から外し ***button-Count*** に 0 を設定します。

関連項目 ***THWMouse::present***

TInputLine DIALOGS.H



TInputLine オブジェクトは，基本的な機能を備えた入力行の文字列エディタです。キーボード入力，マウスクリック，ブロックマークを付けるためのマウスドラッグなど各種の行編集処理を行ないます(***TInputLine::handleEvent*** を参照)。テキストを選択しておいてテキストの入力を行なうと，選択されていたテキストは削除され，その部分に入力されたテキストが入ります。***maxLen*** が x 方向の幅(***size.x***)よりも大きい場合には，水平方向のスクロールがサポートされます。スクロールの指示は，左右にある矢印で行ないます。

メンバ関数 ***getData***，***setData*** は，文字列データ（データメンバ ***data*** によって参照される)を指定されたレコードとの間で読み書きするためのものです。***TInputLine::setState*** は，状態が ***sfActive*** や ***sfSelected*** の状態になったり，逆にそうでなくなったときに，色をそれに合わせてビューを簡単に再表示できるようにするための関数です。

入力行には，***TLabel*** や ***THistory*** オブジェクトがリンクされ

ることがよくあります。

TInputLine は，文字列以外のデータ型を処理するように拡張することもできます。拡張する場合，通常データメンバを追加するとともに，メンバ関数 ***store***，***valid***，***dataSize***，***getData***，および ***setData*** をオーバーライドします。たとえば数値の入力行を定義したい場合，とり得る値の最大値，最小値を持たせ，***valid*** 関数で検査を行なうことになるでしょう。最大値と最小値のデータメンバは，ストリームとの間でロードしたり（ロードコンストラクタを使用して），***store*** したりすることになり，***valid*** は入力が数値であり有効範囲内であることを確認するように変更されることになるでしょう。また ***dataSize*** は，新しい有効範囲のデータメンバのサイズ（おそらく ***sizeof(long)***）も扱うようにすることになるでしょう。不思議に思うでしょうがこの場合，数値自身を格納するデータメンバを追加する必要はありません。数値も文字列値として格納し（文字列なら ***TInputLine*** で処理できる），文字列から数値へ，あるいは数値から文字列への変換を，それぞれ ***getData*** と ***setData*** で行なうようにすることができるからです。

データメンバ

curPos int curPos ;

挿入点（現在のカーソル位置）を示すインデックス。

関連項目 ***TInputLine::selectAll***

data char *data ;

編集データの入っている文字列。

firstPos int firstPos ;

最初に表示される文字を示すインデックス。

関連項目 ***TInputLine::selectAll***

maxLen int maxLen ;

文字列がとり得る最大長（終端の 0 は除く）。

関連項目 ***TInputLine::dataSize***

selEnd int selEnd ;

選択領域の最後（ブロックマークの付いた最後の文字）を示すインデックス。

関連項目 ***TInputLine::selectAll***

selStart int selStart ;

選択領域の最初（ブロックマークの付いた最初の文字）を示すインデックス。

関連項目 ***TInputLine::selectAll***

メンバ関数

コンストラクタ TInputLine (const TRect& bounds, int aMaxLen) ;

TView (*bounds*) を呼び出し，指定された *bounds* の入力ボックスコントロールを作成した後，***state*** に ***sfCursorVis*** を，***options*** に (***ofSelectable*** | ***ofFirstClick***) を，***maxLen*** に ***aMaxLen*** をそれぞれ設定します。さらに ***aMaxlen*** + 1 バイト分のメモリを獲得してクリアし，データメンバ ***data*** にこのメモリ領域へのポインタを入れます。

コンストラクタ TInputLine (StreamableInit streamableInit) ; **protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TView::TView, sfCursorVis, ofSelectable, ofFirst-Click***

デストラクタ　~InputLine ( ) ;

data のメモリ領域を消去した後, ***~TView*** を呼び出して ***TInputLine*** オブジェクトを消去します。

関連項目　***~TView***

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに, オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***, ***ipstream::readData***

dataSize　virtual ushort dataSize ( ) ;

TInputLine::getData 呼び出し, および ***TInputLine::setData*** 呼び出しで使用するレコードのサイズを返します。デフォルトでは ***maxLen*** + 1 を返します。別のデータ型を処理する下位クラスを定義したい場合には, このメンバ関数をオーバーライドしてください。

関連項目　***TInputLine::getData***, ***TInputLine::setData***

draw　virtual void draw ( ) ;

入力ボックスおよびそのデータを表示します。入力ボックスは, ***sfFocused*** であるかどうか(すなわち入力ボックスがカーソルを持つかどうか) によって色を変えて表示され, また入力文字列が入力ボックスのサイズを越える場合には矢印 (片方向または両方向) が表示されます。選択された (ブロックマークの付いた) 文字は, それに対応するパレットを使用して表示されます。

getData　virtual void getData (void *rec) ;

data 文字列を, そのバイト数分だけ(バイト数は, ***dataSize*** 呼び出しによって得られる) *rec* で指定された配列に書き込みます。いろいろなアプリケーションで ***setData*** とともに使用されます。たと

えば一時的な記憶に使用し，入力文字列を別のビューに渡す場合などに使用されます。文字列以外のデータ型を処理する ***TInputLine*** の下位クラスを定義したい場合には，***getData*** をオーバーライドしてください。また ***TInputLine*** で編集を行なった後，***getData*** で文字列を別のデータ型に変換するようにすることも可能です。

関連項目 ***TInputLine::dataSize***，***TInputLine::setData***

getPalette virtual TPalette& getPalette ( ) const ;

デフォルトパレット文字列 ***cpInputLine***，すなわち "¥x13¥x13¥x14¥x15" を返します。

handleEvent void handleEvent (TEvent& event) ;

TView::handleEvent を呼び出した後，入力ボックスが選択されている場合に，すべてのマウスイベントおよびキーボードイベントの処理を行ないます。入力ボックスの標準的な編集機能を実現します。

編集機能としては以下の機能があります。マウスのクリックおよびドラッグによるブロックマーク付け。ブロック削除。挿入または上書きの制御(カーソルの形は自動的に変わる)。必要に応じた自動または手動のスクロール（***data*** 文字列の ***size.x*** に対する相対的なサイズに依る)。 矢印アイコンをマウスクリックすることによる水平方向のスクロール。矢印キー，***HOME*** キー，および ***HELP*** キー（およびこれらのキーと同等の機能の標準 ***CTRL*** キー）による手動のカーソル移動。***DEL*** キー，***CTRL+G*** キーによる文字やブロックの削除。入力ボックスは必要に応じて再表示され，***TInputLine*** の各データメンバもそれに応じて変更されます。

関連項目 ***sfCursorIns***，***TView::handleEvent***，***TInputLine::selectAll***

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***，***TStreamable***，***ipstream***

selectAll void selectAll (Boolean enable) ;

curPos，***firstPos*** および ***selStart*** を 0 にします。*enable* が True の場合には，***selEnd*** に ***data*** 文字列の長さを設定します。したがって，入力行全体が選択されます。*enable* が *False* の場合には ***selEnd*** に 0 を設定します。そのため入力行全体が非選択となります。最後に ***drawView*** を呼び出して，入力ボックスの再表示を行ないます。

関連項目　***TView::drawView***

setData　virtual void setData（void *rec）；

デフォルト処理として，***dataSize*** が返したバイト数分だけ *rec* 配列から ***data*** 文字列にコピーした後，***selectAll***（*True*）を呼び出します。***curPos***，***firstPos*** および ***selStart*** は，0 に設定されます。最後に，***drawView*** を呼び出して入力ボックスの再表示を行ないます。

文字列以外のデータ型を処理する下位クラスを定義したい場合には，***setData*** をオーバーライドしてください。また，***TInputLine*** で編集を行なえるように，***setData*** で別のデータ型を文字列に変換するようにすることも可能です。

関連項目　***TInputLine::dataSize***，***TInputLine::getData***，***TInputLine::selectAll***

setState　virtual void setState（ushort aState, Boolean enable）；

state の変化に応じて，入力ボックスを再表示する必要が生じた場合（たとえばパレットが変更された場合など）に呼び出されます。***TView::setState*** を呼び出し，入力ボックスの ***state*** を指定された *aState* ビットに応じてセットまたはクリアします。さらに *aState* が ***sfSelected*** なら（あるいは *aState* が ***sfActive*** で，かつ入力ボックスが ***sfSelected*** なら），***selectAll***（*enable*）を呼び出します（***selectAll*** がさらに ***drawView*** を呼び出す）。

関連項目　***TView::setState***，***TInputLine::selectAll***

write　virtual void write（opstream& os）；

出力ストリーム *os* への書き込みを行ないます。

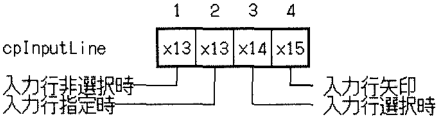
関連項目　***TStreamableClass*, *TStreamable*, *opstream***

関連する関数

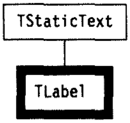
演算子関数で ***TInputLine*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

入力行はデフォルトパレット ***cpInputLine*** を使用し，標準ダイアログパレットのエントリ 19～21 に割り付けます。



TLabel　DIALOGS.H



TLabel オブジェクトは，マウスクリックやカーソルキーや ***GRPH*＋文字**のホットキーによって選択する(ハイライトする)ことのできるビュー内の個別的なテキストです。ラベルは通常，***TView*** へのポインタ（すなわち**リンク**）を介して入力行やクラスタやリストビュアーなどの他のコントロールビューに｢貼り付けられ｣，コントロールの内容を表わします。ラベルを選択する（すなわち「押下する」）ことにより，リンクされているコントロールが選択されま

す。逆にリンクされているコントロールが選択されると，ラベルはハイライト表示にされます。

データメンバ

light Boolean light ;

True であれば，そのラベルとそれにリンクしているコントロールが選択されたことを意味し，ハイライト表示されることを表わします。選択されていない場合には，***light*** は *False* にされます。

link TView *link ;

このラベルにリンクしている ***TView*** コントロールへのポインタ。

メンバ関数

コンストラクタ TLabel (const TRect& bounds, const char *aText, TView *aLink) ;

TStaticText (*bounds, aText*) を呼び出し，指定されたサイズとテキストを備えた ***TLabel*** オブジェクトを作成した後，データメンバ ***link*** に，連結させたいコントロールへのリンク *aLink* を設定します（コントロールを使わない場合には，*aLink* は 0 にする)。***options*** データメンバには，***ofPreProcess*** と ***ofPostProcess*** をセットします。***eventMask*** には，***evBroadcast*** を設定します。*aText* では，ホットキー文字を指定することができます。ホットキー文字は，チルダ（~）で囲んで指定します。

コンストラクタ TLabel (StreamableInit streamableInit) ; **protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを

割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては，第8章も参照してください。

関連項目　***TStaticText::TStaticText***

build　static TStreamable *build ( );

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

draw　virtual void draw ( );

デフォルトパレットの中のしかるべき色を使用してラベルを表示します。

getPalette　virtual TPalette& getPalette ( )　const;

デフォルトパレット文字列 ***cpLabel***，すなわち "¥x07¥x08¥x09¥x09" を返します。

handleEvent　virtual void handleEvent (TEvent& event);

TStaticText::handleEvent を呼び出し，すべてのイベントを処理します。***evMouseDown*** イベントまたはホットキーイベントを受け取った場合，そのラベルとリンクしているコントロールがあればそのコントロールを Link－>Select によって選択します。また ***handleEvent*** は，リンクしているコントロールが発行するブロードキャストイベント ***cmReceivedFocus*** および ***cmReleasedFocus*** の処理も行ない，データメンバ ***light*** の値の再設定を行ないます。必要ならラベルの再表示を行ないます。

関連項目　***TView::handleEvent***，***cmXXXX*** コマンド定数

read　virtual void *read (ipstream& is);

入力ストリーム ***is*** からの読み出しを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream***

shutDown virtual void shutDown ( ) ;

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するために，内部的に使用します。***shutDown*** は，***destroy*** が呼び出された際に，関連するデータメンバの設定を適切に行なえるようにするために多くのクラスでオーバーライドされます。

関連項目 第 6 章『**信頼性の高いプログラムを開発するには**』

write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

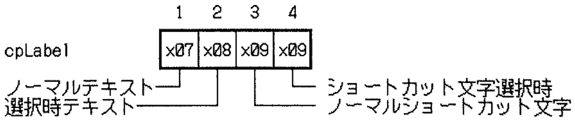
関連項目 ***TStreamableClass***，***TStreamable***，***opstream***

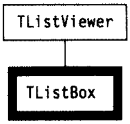
関連する関数

演算子関数で ***TLabel*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

ラベルはデフォルトパレット ***cpLabel*** を使用し，標準ダイアログパレットのエントリ 8～9 に割り付けます。





TListBox は ***TListViewer*** から導出され，ごく一般的に使用されるリストボックス，つまりファイル名などの文字列を要素とするコレクションを表示するボックスの作成を容易にします。***TListBox*** オブジェクトとは，そのような要素からなるリストを１段以上の段数で表示したもののことであり，垂直スクロールバーを持たせることもできます。***TListViewer*** にある水平スクロールバーはサポートされません。***TListViewer*** から継承したメンバ関数により，マウスやキーボードからのカーソル操作による要素の選択（およびハイライト）を行なうことができます。***TListBox*** では，***TListViewer::handleEvent*** も ***TListViewer::draw*** もオーバーライドしていません。したがって，アプリケーションで ***TListBox*** を使用する場合には，まずそれらの関数の説明を参照してください。

TListBox では，***TListViewer*** にはない ***items*** と呼ばれる（プライベートの）データメンバが追加されています。***items*** は，要素をリスト表示したり選択したりする機能を持つ ***TCollection*** オブジェクトへのポインタです。パブリックのメンバ関数である ***list*** は，***items*** ポインタを返します。***TCollection*** オブジェクトへのデータの挿入や，要素が選択されたときに行なう処理は，アプリケーション側で指定しなければなりません。

TListViewer は，デストラクタを ***TView*** から継承します。したがって，***items*** に対する処理を終了した時点で ***items*** の内容を廃棄する処理も，アプリケーション側で指定する必要があります。***newList*** 呼び出しによって，それまでのリストは廃棄されます。したがって，***newList*** (0) 呼び出しを行なってからリストボックスの廃棄を行なえば，すべてのものが解放されることになります。

データメンバ

items　TCollection　*items;　**private**

items は，要素をスクロールさせることのできるコレクションへのポインタです。このコレクションは通常，各要素のテキストとなる文字列を集めたコレクションです。このプライベートメンバは，ユーザからは関数 ***list*** を呼び出すことによってしかアクセスできません。

関連項目　***TListBox::list***

メンバ関数

コンストラクタ　TListBox(const TRect& bounds, ushort aNumCols,
　　TScrollBar　*aScrollBar)；

指定されたサイズ，段数，および *aScrollBar* ポインタで指定される垂直スクロールバーを持つリストボックスコントロールを作成します。***TListViewer*** (*bounds, aNumCols, 0, aScrollBar*) を呼び出します。したがって，水平スクロールバーは付加されません。

list データメンバは最初は空のコレクションであり，継承した ***range*** データメンバには 0 がセットされます。アプリケーションで，リストにされる文字列（またはその他のオブジェクト）を格納する適当な ***TCollection*** を用意する必要があります。list データメンバには，***newList*** を使用してこのコレクションへのポインタを設定する必要があります。

コンストラクタ　TListBox (StreamableInit streamableInit)；　**protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があ

ります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第8章も参照してください。

関連項目　***TListViewer*** コントラクタ，***TListBox::newList***

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうとき，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

dataSize　virtual ushort dataSize ( ) ;

getData および ***setData*** に渡されるレコードで読み書きされるデータのサイズを返します。これら3つのメンバ関数は，グループを初期化するときに便利な関数です。***TListBox::dataSize*** は，デフォルト処理として ***&TCollection*** のサイズ＋***ushort*** のサイズ（それぞれ ***items*** と選択されている要素に対応する）を返します。アプリケーションによっては，このメンバ関数をオーバーライドしなければならない場合もあります。

関連項目　***TListBox::getData***，***TListBox::setData***

getData　virtual void getData (void *rec) ;

TListBox オブジェクトのデータをターゲットのレコードに書き込みます。***getData*** は，デフォルト処理として現在指定されている ***items*** とフォーカスのあるデータメンバを *rec* に書き込みます。アプリケーションによっては，このメンバ関数をオーバーライドしなければならない場合もあります。

関連項目　***TListBox::dataSize***，***TListBox::setData***

getText　virtual void getText (char *dest, short item, short maxLen) ;

この関数を呼び出した ***TListBox*** オブジェクトから文字列を取り出して *dest* に入れます。デフォルト処理では，(***char*** *) ((list ())－>***at*** (*item*)) として ***TCollection*** 中の *item* 番目の要素から得られた文字列を返します。***list*** が文字列以外のオブジェクトか

TL
Std class

らなるコレクションを返してくる場合には，このメンバ関数をオーバーライドする必要があります。***list*** が 0 返した場合には，***getText*** は *dest* に " " （空文字列）を入れます。

関連項目　***TCollection::at***

list　TCollection　*list ( ) ;

list は，プライベートの ***items*** ポインタを返します。

関連項目　***TListBox::items***

newList　virtual void newList (TCollection　*aList) ;

現在のリストを消去し，指定される *aList* と入れ換えることによって新しいリストを作成します。

read　virtual void　*read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream***

setData　virtual void setData (void　*rec) ;

指定された *rec* 配列から ***items*** および ***focused*** の値を読み出し，現在のリストとします。***setData*** は，***newList*** を呼び出すことにより，新しいリストを表示においてしかるべき要素にフォーカスがくるようにします。***getData*** および ***dataSize*** の場合と同様，アプリケーションによってはこのメンバ関数をオーバーライドしなければならない場合もあります。

関連項目　***TListBox::dataSize***，***TListBox::getData***，***TListBox::newList***

write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

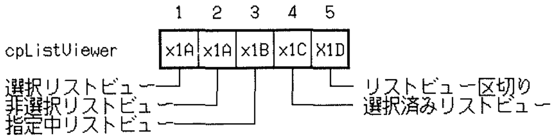
関連項目　***TStreamableClass***，***TStreamable***，***opstream***

関連する関数

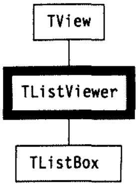
演算子関数で ***TListBox*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

リストボックスはデフォルトパレット ***cpListViewer*** を使用し，標準アプリケーションパレットのエントリ 26～29 に割り付けます。



TListViewer VIEWS.H



TListViewer は，***TListBox*** などいろいろな種類のリストビューアーを導出するために使用される抽象クラスです。***TListViewer*** のメンバによって，以下の機能が用意されています。

- 要素（リスト以外）のリンクリストを表示するためのビュー
- 1個または2個のスクロールバーの制御
- リストの x 方向，y 方向の基本的なスクロール
- ストリームとの間の，ビューおよびそれに付属するスクロールバ

ーの読み書き

- マウスまたはキーボードを使用して，リスト上の要素を選択(ハイライト) する機能
- リサイズやスクロールに対応した処理を行なう ***draw*** メンバ関数

TListViewer の ***getText*** は，純粋仮想メンバ関数なので，要素として表示されるテキストを生成したり，処理したりする機能は，プログラマが作成する必要があります。***TListViewer*** 自体には，リストを記憶する機能はありません。配列，リンクリスト，あるいはそれに類するデータ構造をリストにして表示し，スクロールさせたい場合に，この ***TListViewer*** を使用してください。また ***TListBox*** (リストビュアーにコレクションをリンクする機能を持つクラス) などの ***TListViewer*** から導出したクラスを使用することもできます。

データメンバ

focused short focused ;

フォーカスのある要素の要素番号。各要素には，0～***range*** − 1 までの番号が付けられます。***focused*** の初期値は，0 すなわち最初の要素ですが，マウスクリックするか，**スペース**キーで選択して変更することができます。

関連項目 ***TListViewer::range***

hScrollBar TScrollBar *hScrollBar ;

このビューにリンクされる水平スクロールバーへのポインタ。0 の場合は，このビューに水平スクロールバーがないことを意味します。

numCols short numCols ;

リストコントロール中の段数。

range short range ;

リスト中の現在の総要素数。要素には，0～*range*－ 1 の番号が付けられます。

関連項目 ***TListViewer::setRange***

topItem short topItem ;

表示すべき先頭要素の要素番号。この値は，スクロールにより変わっていきます。要素には，0～*range* － 1 の番号が付けられます。***topItem*** の値は，段数，ビューのサイズ，および ***range*** の値によって変わります。

関連項目 ***TListViewer::range***

vScrollBar TScrollBar *vScrollBar ;

このビューにリンクされる垂直スクロールバーへのポインタ。0 の場合には，このビューに垂直スクロールバーがないことを意味します。

メンバ関数

コンストラクタ TListViewer (const TRect& bounds, ushort aNumCols, TScrollBar *aHScrollBar, TScrollBar *aVScrollBar) ;

まず ***TView***(*bounds*)を呼び出して指定のサイズの ***TListViewer*** オブジェクトを生成し初期化します。次に，データメンバ ***numCols*** に *aNumCols* をセットします。***options*** には，(***ofFirstClick*** | ***ofSelectable***) をセットして，このビューを選択するマウスクリックがあれば，そのイベントがまず ***TListViewer::handleEvent*** に渡ってくるようにします。***eventMask*** には，***evBroadcast*** をセットします。***range*** および ***focused*** に初期値として 0 を入れます。垂直スクロールバー，水平スクロールバーへのポインタは，引数 *aVScrollBar*, *aHScrollBar* で指定します。いずれのスクロールバーも，0 をセットすれば付加されません。これら 2 個のポインタ引数は，***vScrollBar***，***hScrollBar*** データメンバに代入されます。

有効なスクロールバーが指定された場合には，スクロールバーのデータメンバ ***pgStep***, ***arStep*** が，***TListViewer*** のサイズおよび段数にしたがって設定されます。たとえば1段の ***TListViewer*** の場合には，デフォルトの垂直方向 ***pgStep*** は ***size.y*** − 1 となり，デフォルトの垂直 ***arStep*** は 1 となります。

コンストラクタ　TListViewer (StreamableInit streamableInit) ;　　protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第8章も参照してください。

関連項目　***TView::TView***，***TScrollBar::setStep***

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

changeBounds　virtual void changeBounds (TRect& bounds) ;

TView::changeBounds (*bounds*) を呼び出し，***TListViewer*** オブジェクトのサイズを変更します。水平スクロールバーを付加している場合には，***setStep*** を使って ***pgStep*** を更新します。

関連項目　***TView::changeBounds***，***TScrollBar::setStep***

draw　virtual void draw ( ) ;

デフォルトパレットを使用し，表示可能な要素ごとに ***getText*** を繰り返し呼び出すことによって ***TListViewer*** オブジェクトを表示します。どの要素が選択され，どの要素にフォーカスがあるか，またそのビューが ***sfActive*** であるかどうかを考慮して表示を行ないます。

関連項目　*TListViewer::getText*

focusItem　virtual void focusItem (short item);

データメンバ ***focused*** に *item* をセットすることによって，指定された要素にフォーカスを持たせます。また，垂直スクロールバーのデータメンバ ***value*** にも *item* をセットし（垂直スクロールバーがある場合），***topItem*** の再設定も行ないます。

関連項目　*TListViewer::isSelected*, *TScrollBar::setValue*

focusItemNum　virtual void focusItemNum (short item);

focusItem が内部的に使用します。データメンバ ***focused*** に *item* をセットすることによって，指定された要素にフォーカスを持たせます。

関連項目　*TListViewer::focusItemNum*

getPalette　virtual TPalette& getPalette ( )　const;

デフォルトの ***TListViewer*** のデフォルトのパレット文字列である ***cpListViewer***，すなわち "¥x1A¥x1A¥x1B¥x1C¥x1D¥" を返します。

getText　virtual void getText (char　*dest, short item, short maxLen) = 0;

これは純粋仮想関数です。導出クラスでは，このメンバを純粋仮想関数として再宣言するか，*item* を要素インデックスとして指定すれば ***maxLen*** を越えない文字列を返してくる関数でオーバーライドする必要があります。***TListViewer::draw*** が，***getText*** を呼び出す必要があるということに注意してください。

関連項目　*TListViewer::draw*

handleEvent　virtual void handleEvent (TEvent& event);

TView::handleEvent (*event*)　を呼び出し，イベント処理を行ないます。マウスクリックやリストの「自動的な」スクロールがあった場合にはフォーカスを位置を変更します。要素は，マウスのダブ

TL
Std class

ルクリックによって選択することができます。キーボードイベントの処理は次のとおりです。**スペース**キーは，現在フォーカスのある要素を選択します。矢印キー，***ROLLUP***，***ROLLDOWN***，***CTRL-ROLLDOWN***，***CTRL-ROLLUP***，***HOME***，および***HELP***キーは，フォーカスを移動させます。スクロールバーからのブロードキャストイベントがあった場合には，フォーカスの位置を変更し，必要に応じてビューの再表示を行ないます。

関連項目　***TView::handleEvent***，***TListViewer::focusItem***

isSelected　virtual Boolean isSelected (short item) ;

指定された *item* が選択されている（フォーカスを得ている）場合，すなわち *item* == ***focused*** であれば *True* を返します。

関連項目　***TListViewer.focusItem***

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream***

selectItem　virtual void selectItem (short item) ;

リストの *item* 番目の要素を選択し，以下の呼び出しによって，このことをグループのオーナーにブロードキャストします。

```
message(owner, evBroadcast, cmListItemSelected, this);
```

関連項目　***TListViewer::focusItem***，***message***

setRange　void setRange (short aRange) ;

データメンバ ***range*** に ***aRange*** をセットします。垂直スクロールバーが付加されている場合には，必要に応じてそのパラメータの再設定を行ないます（再表示が必要な場合には，***TScrollBar::drawView*** の呼び出しも行なわれます)。現在フォーカスのある要素が新しい ***range*** の範囲外となった場合には，データメンバ ***focused*** には0が設定されます。

関連項目　***TListViewer::range***, ***TScrollBar::setParams***

setState　virtual void setState (ushort aState, Boolean enable)；

enable が *True* なら，***TView::setState*** (*aState, enable*) を呼び出し，***TListViewer*** オブジェクトの状態を変更します。したがって *aState* によって，そのビューを表示するようにしたり，表示しないようにしたりすることができます。さらに，*aState* が ***sfSelected*** かつ ***sfActive*** である場合には，スクロールバーの再表示を行ないます。*aState* が ***sfSelected*** であって ***sfActive*** でない場合には，スクロールバーを表示しないようにします。

関連項目　***TView::setState***, ***TScrollBar::show***, ***TScrollBar::hide***

shutDown　virtual void shutDown ( )；

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するために内部的に使用します。***shutDown*** は，***destroy*** が呼び出されたときに，関連するデータメンバの設定を適切に行なえるようにするために多くのクラスでオーバーライドされます。

関連項目　第6章**『信頼性の高いプログラムを開発するには』**

write　virtual void write (opstream& os)；

出力ストリーム *os* への書き込みを行ないます。

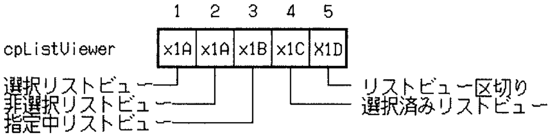
関連項目　***TStreamableClass***, ***TStreamable***, ***opstream***

関連する関数

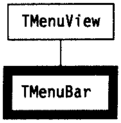
演算子関数で ***TListViewer*** に関係するものもありますが，メンバ関数とはなっていません。詳細については322ページを参照してください。

パレット

リストビューアーは，デフォルトパレット ***cpListViewer*** を使用し，アプリケーションの標準パレットのエントリ 26～29 に割り付けます。



TMenuBar　　　MENUS.H



TMenuBar オブジェクトは，水平方向のメニューバーとなり，以下の操作によってメニュー選択を行なうことができるようにします。

- ■ メニューの直接のクリック
- ■ ***F10*** による選択およびホットキー
- ■ 選択（ハイライト）と**リターン**キー入力
- ■ ホットキー

メインメニューの選択項目は，最上段のメニューバーに表示されます。これは通常，ユーザが作成する ***TApplication*** オブジェクトによって所有される ***TMenuBar*** 型のオブジェクトによって実現されます。サブメニューは，***TMenuBox*** 型のオブジェクト内に表示されます。***TMenuBar*** も ***TMenuBox*** も，***TMenuView*** から導出されます（***TMenuView*** は，***TView*** から導出される)。

Turbo Vision を使用した，たいていのアプリケーションにおいては，ユーザが直接メニューオブジェクトを扱うことはないでしょう。new TMenuItem および new TMenu 呼び出しを適当な回数

だけネストさせて ***TApplication::initMenuBar*** をオーバーライドすれば，後は Turbo Vision の側で標準的なメニュー処理を行なってくれます。

メンバ関数

コンストラクタ　TMenuBar (const TRect& bounds, TMenu *aMenu) ;

TMenuView (***bounds***) を呼び出し，メニューバーを作成します。グローモードには，***gfGrowHiX*** をセットします。データメンバ ***options*** には ***ofPreProcess*** をセットし，ホットキーが機能するようにします。またデータメンバ ***menu*** に ***aMenu*** をセットし，メニューの選択項目を指定します。

コンストラクタ　TMenuBar (StreamableInit streamableInit) ;　**protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TMenuView::TMenuView***，***gfXXXX*** グローモードフラグ，***ofXXXX*** オプションフラグ，***TMenuView::menu***, ***TMenu***

build　static TStreamable *build () ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***, ***TStreamable***

draw　virtual void draw () ;

デフォルトパレットを使ってメニューバーを表示します。***menu*** リ

ンクリストにつながっている各 ***TMenuItem*** のデータメンバ ***name*** および ***disabled*** を読み出し，メニューの見出しを適当な色で表示します。***current*** な（選択されている）項目は，ハイライト表示にします。

getItemRect virtual TRect getItemRect (TMenuItem *item) ;

TMenuView::getItemRect をオーバーライドします。指定されたメニュー項目が占める矩形領域を返します。***mouseInView*** と一緒に使用することで，指定のメニュー項目上でマウスクリックがあったかどうかを判断することができます。

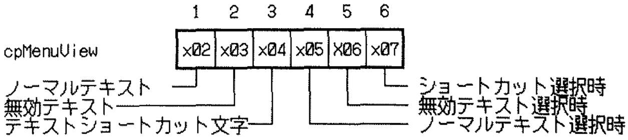
関連項目 ***TMenuView::getItemRect***, ***TView::mouseInView***

関連する関数

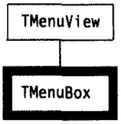
演算子関数で ***TMenuBar*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

メニューバーは，他のすべてのメニュー関係のビューと同様にデフォルトパレット ***cpMenuView*** を使用し，標準的なアプリケーションパレットのエントリ 2～7 に割り付けます。



TMenuBox　　MENUS.H



TMenuBox オブジェクトは，垂直方向のメニューボックスとなります。メニューボックスには処理項目からなる任意のリストを持たせることができ，そこから項目を選択することができます。項目をサブメニューにすることもできます。メニューバーの場合と同様，色付けによって無効な項目であることを示すことができます。メニューボックスは，メニューバーのサブメニュー，または別のメニューボックスのサブメニューとしてインスタンス化することができます。また，単体でポップアップメニューとしても使用できます。

メンバ関数

コンストラクタ　TMenuBox (const TRect& bounds, TMenu *aMenu) ;
TMenuBox (const TRect& bounds,TMenu *aMenu,
TMenuView *aParentMenu =0) ;

TMenuView (*bounds*) を呼び出して ***TMenuBox*** オブジェクトを作成した後，*aMenu* 中の各項目の幅および長さに合わせて *bounds* パラメータを再設定します。

options データメンバの ***ofPreProcess*** ビットを立て，ホットキーが機能するようにします。***state*** の ***sfShadow*** を立てます。データメンバ ***menu*** には，メニュー選択項目となる *aMenu* をセットします。コンストラクタの第 2 の形式では，***parentMenu*** にセットされる引数として明示的に *aParentMenu* をとることができます（デフォルト値は 0)。

コンストラクタ　TMenuBox (StreamableInit streamableInit) ;　　**protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第8章も参照してください。

関連項目　***TMenuView::TMenuView***，***sfXXXX*** 状態フラグ，***ofXXXX*** オプションフラグ，***TMenuView::menu***，***TMenuView::parentMenu***

build　static TStreamable *build ( )；

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

draw　virtual void draw ( )；

フレーム付きのメニューボックスおよびそれにリンクされるメニュー項目をデフォルト色で表示します。

getItemRect　virtual TRect getItemRect (TMenuItem *item)；

TMenuView::getItemRect をオーバーライドします。指定されたメニュー項目が占める矩形領域を返します。マウスクリックが指定したメニュー選択項目上で起こったのかどうかを判断するために使用することができます。

関連項目　***TMenuView::getItemRect***，***TView::mouseInView***

read　virtual void *read (ipstream& is)；

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream***

write　virtual void write (opstream& os)；

出力ストリーム *os* への書き込みを行ないます。

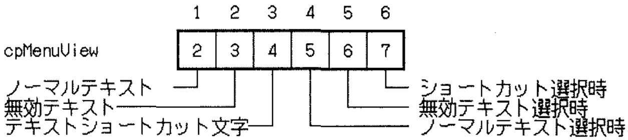
関連項目 *TStreamableClass*, *TStreamable*, *opstream*

関連する関数

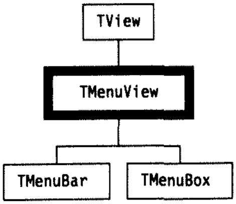
演算子関数で ***TMenuBox*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

メニューボックスは，他のすべてのメニュー関係のビューと同様にデフォルトパレット ***cpMenuView*** を使用し，標準アプリケーションパレットのエントリ 2~7 に割り付けます。



TMenuView　　　MENUS.H



TMenuView は，メニューバーやメニューボックス(プルダウンまたはポップアップ) を導出するための抽象基本クラスです。***TMenuView*** 自体をインスタンス化することはできません。

データメンバ

current TMenuItem *current;

現在選択されているメニュー項目へのポインタ。

menu TMenu *menu;

このメニューで使用される ***TMenu*** オブジェクト(メニュー項目のリンクリスト) へのポインタ。***menu*** ポインタを参照することによって，このメニュービューで使用されるすべてのメニュー項目のデータメンバにアクセスすることができます。

関連項目 ***TMenuView::findItem***, ***TMenuView::getItemRect***, ***TMenu struct***

parentMenu TMenuView *parentMenu;

このメニューのオーナーである ***TMenuView*** オブジェクト（または ***TMenuView*** から導出される任意のクラス）へのポインタ。***TMenuView*** はグループではありません。ここでの所有関係の考え方は ***TGroup*** の場合の所有関係よりもずっと単純であり，メニューをネストさせることができます。つまりサブメニューの選択を行ない，また「親」メニューに戻ってくることができます。たとえばメニューバーで選択を行なうと，サブメニューが「プルダウン」されることになりますが，この場合メニューバーはメニューボックスの親メニューです。

関連項目 ***TMenuBox::TMenuBox***

メンバ関数

コンストラクタ TMenuView (const TRect& bounds);

TView::TView (***bounds***) を呼び出し，サイズが ***bounds*** の

TMenuView オブジェクトを作成します。***current*** な ***TMenuItem***, ***parentMenu***, および ***menu*** ポインタには，0 がセットされます。デフォルトの ***eventMask*** には，***evBroadcast*** がセットされます。この ***TMenuView*** コンストラクタは，導出クラスである ***TMenuBar*** および ***TMenuBox*** から呼び出されます。直接呼び出されることは，めったにありません。

コンストラクタ　TMenuView（StreamableInit streamableInit）;　　protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TView::TView***, ***evBroadcast***, ***TMenuBar::TMenuBar***, ***TMenuBox::TMenuBox***

TM
Std class

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***, ***ipstream::readData***

execute　virtual ushort execute ( ) ;

ユーザがメニュー項目を選択するかキャンセルするまで，メニュービューを実行します。選択されたメニュー項目に割り当てられているコマンドを返します。メニューがキャンセルされた場合には 0 を返します。このメンバ関数は，***execView*** 以外から呼び出してはなりません。

関連項目　***TGroup::execView***

findItem　TMenuItem *findItem (char ch) ;

ホットキー（ハイライトしている文字）が ***toupper*** (*ch*) であるメニュー項目へのポインタを返します。そのようなメニュー項目がな

い場合，またはそのメニュー項目が無効になっている場合には，0 を返します。*ch* の大文字・小文字の区別は無視されます。

getHelpCtx virtual ushort getHelpCtx ( ) ;

デフォルト処理として，現在有効なメニュー選択項目のヘルプコンテキストを返します。ヘルプコンテキストが ***hcNoContext*** である場合には，親のメニューの現在のコンテキストを探します。親のメニューがない場合には，***getHelpCtx*** は ***hcNoContext*** を返します。

関連項目　***hcXXXX*** ヘルプコンテキスト定数

getItemRect virtual TRect getItemRect (TMenuItem *item) ;

TMenuView から導出されたクラスでは，このメンバ関数をオーバーライドして，マウスイベントを処理するようにする**必要があります**。導出クラスでオーバーライドした関数は，指定したメニュー項目が占める矩形領域を返すようにしなければなりません。***mouseInView*** と一緒に使用することによって，マウスクリックが指定したメニュー選択項目上で行なわれたものであるかどうかを確かめ，それに基づいて必要な処理を決定することができます。

関連項目　***TMenuBar::getItemRect***，***TMenuBox::getItemRect***，***TView::mouseInView***

getPalette virtual TPalette& getPalette ( ) const ;

デフォルトパレット文字列 ***cpMenuView***，すなわち "¥x02¥x03¥x04¥x05¥x06¥x07" を返します。

handleEvent virtual void handleEvent (TEvent& event) ;

メニューイベントを処理する必要がある場合に呼び出されます。マウスまたはキーボード（ホットキーも含めて）によってどのメニュー項目が選択されたかを調べ，それに対応したコマンドイベントを ***putEvent*** を使用して発生します。

関連項目　***TView::handleEvent***，***TView::putEvent***，***TMenuView::hotKey***

hotKey TMenuItem *hotKey (ushort keyCode) ;

keyCode で指定されたホットキーがリンクされているメニュー項目へのポインタを返します。そのようなメニュー項目が存在しない場合，またはその項目が無効になっている場合には，0 を返します。ホットキーは，***TProgram::initMenuBar*** で指定しますが，通常，ファンクションキーか，**GRPH** キーと何らかの文字との組み合わせを使用します。***hotKey*** は，キー入力イベントによってメニュー中の項目が選択されたかどうかを調べるために，***handleEvent*** から呼び出されます。

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***，***TStreamable***，***ipstream***

write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目 ***TStreamableClass***，***TStreamable***，***opstream***

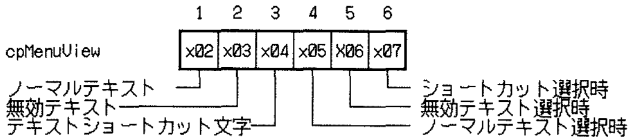
TM
Std class

関連する関数

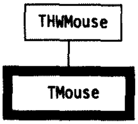
演算子関数で ***TMenuView*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

ニュー関係のビューは，すべてデフォルトパレット ***cpMenuView*** を使用し，標準アプリケーションパレットのエントリ 2~7 に割り付けます。



TMouse　　SYSTEM.H



TMouse は，低レベルのマウス処理関数を用意します。このクラスを含め，SYSTEM.H にあるシステム関係のクラスについては簡単にしか紹介しません。これらのクラスは Turbo Vision が内部的に使用するものであり，通常のアプリケーションであれば直接使用する必要はありません。

メンバ関数

コンストラクタ　TMouse ( ) ;

TMouse::show を呼び出し，マウスカーソルを表示します。

関連項目　***TMouse::show***

デストラクタ　~TMouse ( ) ;

THWMouse::hide を呼び出し，マウスカーソルを表示しないようにします。

関連項目　***THWMouse::hide***

getEvent　static void getEvent (MouseEventType& me) ;

THWMouse::getEvent (me) を呼び出し，***MouseEventType*** 構造体 *me* のデータメンバ ***buttons***，***where.x***，***where.y*** および ***doubleClick*** に値を設定します。

関連項目　***THWMouse::getEvent***，***MouseEventType***

hide　static void hide ( ) ;

THWMouse::hide を呼び出し，マウスカーソルを表示しないようにします。

関連項目　***THWMouse::hide***

present　static Boolean present ( ) ;

THWMouse::present を呼び出します。マウスが有効なら(すなわち **buttonCount** が 0 でなければ) *True* を返します。マウスが無効なら *False* を返します。

関連項目　***THWMouse::present***

registerHandler　static void registerHandler (unsigned mask,
void (far *func)( )) ;

THWMouse::registerHandler(mask, func)を呼び出し，*func* を現在有効なマウスハンドラとして登録します。

関連項目　***THWMouse::registerHandler***

resume　static void resume ( ) ;

し ***THWMouse::resume*** を呼び出します。ハンドラの再登録を行ない，**buttonCount** の再設定を行なうことでマウスの復旧を行ないます。

関連項目　***THWMouse::resume***

setRange　static void setRange (ushort rx, ushort ry) ;

THWMouse::setRange (rx, ry) を呼び出し，指定された引数をマウスの有効範囲に設定します。

関連項目　***THWMouse::setRange***

show　static void show ( ) ;

THWMouse::show を呼び出し，マウスカーソルを表示します。

関連項目　***THWMouse::show***

suspend　static void suspend ( ) ;

THWMouse::suspend を呼び出します。***present*** が *False* を返してきた場合には，何もしません。***present*** が *True* を返してきた場合にはマウスの表示をやめ，ハンドラを登録から外し ***buttonCount*** を 0 にします。

関連項目　***THWMouse::suspend***

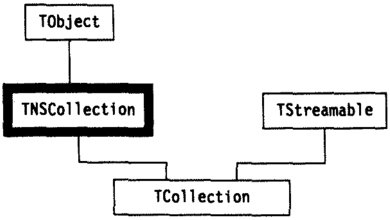
TMouseEventType　　SYSTEM.H

TMouseEventType 構造体はマウスイベント用のデータとして，ボタン番号，ダブルクリックであるかどうか，およびクリックが検出された点を表わす座標を格納します。

```
struct MouseEventType
{
   uchar buttons;
   Boolean doubleClick;
   TPoint where;
};
```

関連項目　***TEvent::getMouseEvent***，***TView::handleEvent***，***THWMouse::getEvent***

TNSCollection　　TVOBJS.H



TNSCollection は，ストリームすることのできないコレクショ

ンです（したがってプレフィクスに *NS* が付いている）。要素として，他のオブジェクトをとることができます。これは，主として，（***TStreamable*** とともに多重継承させることによって）ストリーム入出力可能なコレクションクラスである ***TCollection*** を導出するための基本クラスとするために作られたクラスです。***TNSCollection*** は ***TCollection*** の基本クラスとして，コレクションに対する要素の追加，アクセス，削除を行なう機能を用意しています。一方 ***TStreamable*** は，コレクションとの間で読み書きを行なうストリームに名前を付けたり，そのようなストリームを作成する機能を ***TCollection*** にもたらします。

コレクションというのは，従来の配列，集合，リストといったものより一般的な概念です。***TNSCollection*** オブジェクトは，実行時に動的に自分自身のサイズを変更することができ，***TCollection***, ***TNSSortedCollection***, ***TSortedCollection***, ***TStringCollection*** および ***TResourceCollection*** など，それぞれの用途に応じて特殊化された導出クラスの基本となります。***TNSCollection*** には，要素の追加や削除を行なうメンバ関数以外にも，コレクションの各要素に対して指定した関数を呼び出していくという繰り返し処理ルーチンもいくつか用意されています。

データメンバ

count　ccIndex count ;　protected

コレクション内の現在の要素数です。最大 ***maxCollectionSize*** 個までです。

関連項目　***maxCollectionSize*** 変数

delta　ccIndex delta ;　protected

items のリストが一杯になったとき，リスト拡張を行なうために追加される単位要素数。***delta*** を 0 にすると，コレクションは ***limit*** に設定された要素数以上は拡張できないことになります。

関連項目　***limit***，***TNSCollection*** コンストラクタ

items void * *items ; protected

汎用的な要素ポインタの配列へのポインタ。

limit ccIndex limit ; protected

items リストに現在割り当てられているサイズ（要素数）。

関連項目 ***delta***, ***TNSCollection*** コンストラクタ

limit Boolean shouldDelete ; protected

True の場合（デフォルトは *True*），***TNSCollection*** デストラクタは ***limit*** を 0 にする前に ***freeAll*** を呼び出します。*False* の場合には，デストラクタは ***limit*** に 0 を設定するだけです。

関連項目 ***TNSCollection::TNSCollection***, ***TNSCollection::freeAll***

メンバ関数

TN
Std class

コンストラクタ TNSCollection (ccIndex aLimit, ccIndex aDelta) ;

limit に *aLimit* を，***delta*** に *aDelta* をセットしてコレクションを作成します。***count*** と ***items*** には 0 を入れます。***shouldDelete*** には *True* を設定します。最初は，コレクションの要素数は *aLimit* に制限されますが，メモリを使いきるか，要素数が ***maxCollectionSize*** に達するまで *aDelta* 個ずつ拡張していくことができます。

コンストラクタ TNSCollection ( ) ;

このコンストラクタは ***shouldDelete*** を *True* にし，それ以外のデータメンバにはすべて 0 を設定します。

関連項目 ***TNSCollection::shouldDelete***, ***TNSCollection::count***, ***TNSCollection::limit***, ***TNSCollection::delta***

デストラクタ ~TNSCollection ( ) ;

shouldDelete が *True* なら ***TNSCollection::freeAll*** を呼び出

し，コレクションの全要素を削除・消去し，***limit*** に 0 を設定します。***shouldDelete*** が *False* なら ***limit*** に 0 を設定しますが，コレクションの消去は行ないません。

関連項目　***TNSCollection::shouldDelete***，***TNSCollection::freeAll***，***TNSCollection::setLimit***

at　void *at (ccIndex index) ;

index で指定されたコレクション中の要素へのポインタを返します。このメンバ関数によって，コレクションを配列として扱うことができます。*index* が 0 以下または ***count*** 以上である場合には，***coIndexError*** を引数にして ***error*** メンバ関数を呼び出した後 0 を返します。

関連項目　***TNSCollection::indexOf***

atInsert　void atInsert (ccIndex index, void *item) ;

index 番目以降の要素を 1 つずつ後ろにずらし，*index* 番目の位置に *item* を挿入します。*index* が 0 または ***count*** 以上である場合には，***coIndexError*** を引数にして ***error*** メンバ関数を呼び出し，新しい *item* の挿入は行ないません。***atInsert*** 呼び出し以前に ***count*** が ***limit*** となっていた場合には，***setLimit*** 呼び出しを行ないコレクションの割り当てサイズを *delta* 個分だけ拡張します。***setLimit*** 呼び出しでコレクションを拡張できなかった場合には，***coOverflow*** を引数にして ***error*** メンバ関数を呼び出し，新しい *item* の挿入は行ないません。

関連項目　***TNSCollection::at***，***TNSCollection::atPut***

atPut　void atPut (ccIndex index, void *item) ;

index 番目の位置の要素を，指定した *item* で入れ換えます。*index* が 0 または ***count*** 以上である場合には，***coIndexError*** を引数にして ***error*** メンバ関数を呼び出します。

関連項目　***TNSCollection::at***，***TNSCollection::atInsert***

atRemove　void atRemove (ccIndex index) ;

index 番目の位置にある要素を削除し，それ以降の要素を 1 つずつ前にずらしていきます。***item*** 自体は消去されません。***count*** は 1 減らしますが，コレクションに割り当てられているメモリ（ ***limit*** で与えられる）を減らすわけではありません。*index* が 0 または ***count*** 以上である場合には，***error***(1,0)を呼び出します。***atRemove*** の動作と ***atFree*** の動作を比較してみてください。***atFree*** は ***atRemove*** を実行した後，***delete*** (*item*)としてその要素を消去します。

関連項目　***TNSCollection::atFree***, ***TNSCollection::remove***

error　static void error (ccIndex code, ccIndex info) ;

コレクションにエラーが発生した場合に必ず呼び出されます。デフォルト処理として，(212 − *code*)の実行時エラーを発生します。

firstThat　void *firstThat (ccTestFunc Test, void *arg) ;

firstThat は，***Boolean*** 関数 **Test* に *arg* で指定される引数リスト（空でもかまわない）を渡し，コレクション内の各要素に適用していきます。**Test* が *True* を返してくるまで続けられます。戻り値は，**Test* が *True* を返した要素へのポインタです。**Test* がすべての要素について *False* を返してきた場合には，戻り値は 0です。***ccTestFunc*** は，以下のように typedef 宣言されています。

```
typedef Boolean (*ccTestFunc) (void *, void *);
```

Test* の 1 番目の引数には，コレクションの要素へのポインタが入れられていきます。Test* の 2 番目の引数には，***firstThat*** の引数であるポインタ *arg* が設定されます。***firstThat*** は以下のようなつくりになっています。

```
void *TNSCollection::firstThat( ccTestFunc Test, void *arg )
{
   for( ccIndex i = 0; i < count; i++ )
      {
      if( Test( items[i], arg ) == True )
         return items[i];
      }
   return 0;
}
```

TN
Std class

protected データメンバ ***items*** は void **型なので，items [i] は void * 型です。

関連項目 ***TNSCollection::lastThat, TNSCollection::forEach***

forEach void forEach (ccAppFunc action, void *arg) ;

繰り返し処理関数 ***forEach*** は，関数 ****action*** で指定された処理をコレクションの各要素に適用していきます。 *arg* ポインタによって，処理関数に別途引数を渡すことができます。***ccAppFunc*** は，以下のように typedef されています。

```
typedef void (*ccAppFunc) ( void *, void *);
```

****action*** の1番目の引数には，コレクションの要素へのポインタが入れられていきます。****action*** の2番目の引数には，***forEach*** の引数であるポインタ *arg* が設定されます。***forEach*** は以下のようなつくりになっています。

```
void TNSCollection::forEach( ccAppFunc action, void *arg )
{
   for( ccIndex i = 0; i < count; i++ )
      action( items[i], arg );
}
```

protected データメンバ ***items*** は void**型なので，items [i] は void *型です。

関連項目 ***TNSCollection::firstThat, TNSCollection::lastThat***

free void free (void *item) ;

指定の *item* を削除・消去します。以下と同等の機能を果たします。

```
remove( item );
delete( item );
```

関連項目 ***TNSCollection::remove***

freeAll void freeAll ( ) ;

コレクションの全要素を削除・消去し，***count*** に 0 を設定します。

関連項目　***TNSCollection::removeAll***

indexOf　virtual ccIndex indexOf (void *item) ;

指定された *item* のインデックスを返します。つまり，***TNSCollection::at*** と逆の働きをします。*item* がコレクション内に見つからなかった場合には，***error***(1,0)を呼び出します。

関連項目　***TNSCollection::at***

insert　virtual void insert (void *item) ;

item をコレクションに挿入し,必要なら他のインデックスを調整します。デフォルト処理として，***atInsert*** (*count*, *item*)を呼び出すことにより，コレクションの最後に挿入を行ないます。

関連項目　***TNSCollection::atInsert***

lastThat　void *lastThat (ccTestFunc Test, void *arg) ;

lastThat は, Boolean 関数 **Test* に *arg* で指定される引数リスト（空でもかまわない）を渡し，コレクションの最終要素から遡りながら各要素に適用していきます。**Test* が *True* を返してくるまで続けられます。戻り値は，**Test* が *True* を返した要素へのポインタです。**Test* が，すべての要素について *False* を返してきた場合には，戻り値は 0 です。***ccTestFunc*** は，以下のように typedef されています。

```
typedef Boolean (*ccTestFunc) (void *, void *);
```

Test の 1 番目の引数には，コレクションの要素へのポインタが入れられていきます。*Test* の 2 番目の引数には, ***lastThat*** の引数であるポインタ *arg* が設定されます。***lastThat*** は以下のようなつくりになっています。

```
void *TNSCollection::lastThat( ccTestFunc Test, void *arg )
{
   for( ccIndex i = count; i > 0; i-- )
      {
      if( Test( items[i-1], arg ) == True )
          return items[i-1];
      }
   return 0;
}
```

protected データメンバ ***items*** は void** 型なので，items [i] は void* 型です。

関連項目　***TNSCollection::firstThat***，***TNSCollection::forEach***

pack　void pack ( ) ;

コレクション中のすべてのヌルポインタを削除し，要素をずらして隙間（ギャップ）を埋めていきます。

関連項目　***TNSCollection::remove***，***TNSCollection::removeAll***

remove　void remove (void *item) ;

item で指定された要素をコレクションから削除します。***atRemove*** (***indexOf*** (*item*))と同等の機能です。

関連項目　***TNSCollection::atRemove***，***TNSCollection::indexOf***

removeAll　void removeAll ( ) ;

count を 0 にし，コレクションの全要素を削除します。

関連項目　***TNSCollection::remove***，***TNSCollection::atRemove***

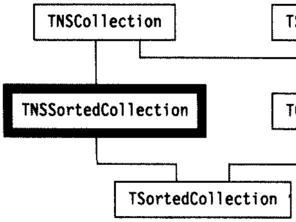
setLimit　virtual void setLimit (ccIndex aLimit) ;

コレクションの割り当てサイズを *aLimit* に変更することで，コレクションを拡張または縮小します。*aLimit* が ***count*** よりも小さい場合には ***count*** が *aLimit* にセットされ，*aLimit* が ***maxCol-***

lectionSize よりも大きい場合には ***maxCollectionSize*** が *aLimit* にセットされます。そして *aLimit* が現在の ***limit*** と異なる場合には，*aLimit* 個の要素からなる新しい ***items*** 配列を割り当て，古い ***items*** 配列の内容を新しい配列にコピーしてから，古い配列を消去します。

関連項目　***TNSCollection::limit***, ***TNSCollection::count***, ***maxCollectionSize*** 変数

TNSSortedCollection　　TVOBJS.H



抽象クラス ***TNSSortedCollection*** は，***TNSCollection*** を特殊化した導出クラスであり，指定したキー（キーの重複を許すこともできる）によってソートされるコレクションです。ストリームすることはできません。***TNSSortedCollection*** のインスタンスをを作ることはできません。他の標準クラスを導出したり，ユーザが導出クラスを定義するときのベースとするためだけに存在します。

ソートには，純粋仮想（かつ private）のメンバ関数 ***compare*** を使用します。この関数はオーバーライドする（あるいは純粋仮想として再定義する）**必要があります**。実際には，導出してきたクラスでこの関数をオーバーライドし，それぞれの用途に合わせてコレクションをソートする機能を持たせることになります。新しい要素を追加するときには，自動的に ***compare*** が指定する順序にしたがって挿入を行ないます。要素の配置には，バイナリの（デフォルトがバイナリ）***search*** 関数（これも仮想）を使用することもできます。***compare*** で使用されるポインタを返してくる関数 ***indexOf*** も仮想となっており，***compare*** がそれ以外の情報を必要とするような場

TN
Std class

合のために，オーバーライドすることができるようになっています。

ストリーム入出力可能なソート済みコレクションとしたい場合には，***TSortedCollection*** を使用する必要があります。これは，***TNSSortedCollection*** と ***TCollection***（これの基本は ***TStreamable***）を多重継承したクラスです。ストリーム入出力可能かどうかという点を除けば，***TSortedCollection*** と ***TNSSortedCollection*** の機能は同じです。

データメンバ

duplicates Boolean duplicates ;

インデックスを重複させることができる場合に *True* にします。できない場合には *False* にします。デフォルトは *False* です。***duplicate*** が *True* の場合には，***search***, ***insert***, および ***indexOf*** の各メンバ機能の動作が異なってきます（各メンバ関数の項を参照)。

メンバ関数

コンストラクタ TNSSortedCollection (ccIndex aLimit, ccIndex aDelta) ;

TCollection コンストラクタを呼び出し，***count***, ***items***, および ***limit*** に 0 をセットします。さらに ***setLimit*** (*aLimit*)を呼び出してコレクションのサイズを *aLimit* とし，***delta*** に *aDelta* をセットします。なお，***ccIndex*** は ***int*** として typedef されています。***duplicates*** には *False* をセットします。キーを重複させてもよい場合には，***duplicates*** に *True* をセットする必要があります。

関連項目 ***TCollection*** コンストラクタ，***TCollection*** データメンバ

compare virtual int compare (void *key1, void *key2) = 0 ; private

compare は純粋仮想関数であり，導出クラスでは必ずオーバーライドしなければなりません。***compare*** は 2 つのキー値を比較し，以下

のような結果を返すようにしなければなりません。

< 0	if	*key1*	<	*key2*
0	if	*key1*	=	*key2*
> 0	if	*key1*	>	*key2*

key1, *key2* は汎用のポインタです。***keyOf*** メンバ関数を使用してコレクション要素から取り出すことができます。***search*** メンバ関数は，この ***compare*** を使用して要素の比較を行ないながらコレクション要素のバイナリサーチを行ないます。

insert virtual void insert (void *item) ;

duplicates が *False* の場合，***insert*** の機能は次のとおりです。指定された要素がソート済みコレクション中に見つからなかった場合，しかるべきインデックス位置にその要素を挿入します。***search*** を呼び出し，その要素が存在するかどうかを調べるとともに，存在しない場合には挿入位置を決定します。***insert*** の定義は，以下のとおりです。

```
{
   ccIndex i;

   if (search(keyOf(item), i) == 0)
       atInsert(i,item);
}
```

duplicates が *True* の場合には，同じキーを持つ要素の一番先頭に（そういう要素がある場合）指定された要素を挿入します。

関連項目 ***TNSSortedCollection::search***, ***TCollection::atInsert***

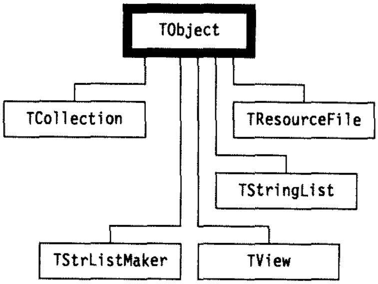
search virtual Boolean search (void *key, ccIndex& index) ;

ソート済みのコレクション中に *key* で指定された要素が見つかった場合には *True* を返します。その場合 *index* にはその見つかった要素のインデックスがセットされます。見つからなかった場合，*index* には，その要素が挿入されるべき位置を示すインデックスが

セットされます。***duplicates*** が *True* のとき，キーの重複があった場合には，***search*** はキーの一致する一番最初の要素を *index* に返します。

関連項目 ***TNSSortedCollection::compare***，***TNSSortedCollection::insert***

TObject　TVOBJS.H



TObject は，数々のクラスからなる Turbo Vision のクラス階層の始点となるものです。***TObject*** には親はありませんが，子孫となるクラスがたくさんあります。***TPoint*** と ***TRect***(および初期化やストリーム管理を行なうためのクラスと構造体) 以外の Turbo Vision 標準のほとんどのクラスは，もとをただせば ***TObject*** から導出されています。

メンバ関数

デストラクタ　virtual ~TObject();

動的オブジェクトの整理 (cleanup) および廃棄を必要に応じて行ないます。

destroy　static void destroy (TObject *ob) ;

TObject から導出された型のオブジェクト (すなわち ***new*** 演算子

を使用して作成されたすべてのオブジェクト) *ob* を消去したい場合に，***destroy*** を呼び出してください。***destroy*** は，オブジェクトの占めていたメモリを正しく解放し，オブジェクトを消去します。以下のように，C++の ***delete*** 演算子のかわりにこの関数を使用してください。

```
TDialog *dlg = new TDialog( ... );
// delete dlg;     // こうしないで、
destroy( dlg );    // こうしてください。
```

関連項目　第 6 章『**信頼性の高いプログラムを開発するには**』

shutDown　virtual void shutDown ( ) ;

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するために内部的に使用します。***destroy*** が呼び出されたとき ***shutDown*** は，関連するデータメンバの設定を適切に行なえるようにするために，多くのクラスでオーバーライドされます。

関連項目　第 6 章『**信頼性の高いプログラムを開発するには**』

TPalette　　VIEWS.H

TPalette は，パレット配列を作成したり操作したりするための単純なクラスであり，プログラマが内部的な構造にまで関与しなくてもよいようにします。パレットは ***char*** 型の配列であり，多くの場合文字列として参照されますが，従来の C にあったヌルで終わる文字列とはなっていません。実際，パレットが 0バイトである場合もあります。したがって，通常の C の文字列関数は使用することができません。パレット文字列の第 1 バイト目には，文字列長が格納されます(第 1 バイト目自身は数えない)。基本的なビューは，それぞれパレット文字列を用意しており，スクロールバーやフレーム，ボ

タン，テキストなど，ビューの各部位に通常割り当てておく色を決めています。詳細については第 4 章『**ビュー**』を参照してください。

メンバ関数

コンストラクタ

TPalette (const char *d, ushort len) ;
TPalette (const TPalette& tp) ;

最初の形式の場合，文字列 *d*，長さ *len* の ***TPalette*** オブジェクトを作成します。プライベートメンバ ***data*** に *len* がセットされ，それ以降のバイトに配列 *d* をセットします。2 番目の形式の場合，パレット *tp* をコピーすることによって新しいパレットを作成します。

デストラクタ

~TPalette ( ) ;

パレットを消去します。

operator =

TPalette& operator = (const TPalette& tp) ;

p = tp; というコードで，パレット *tp* をパレット *p* にコピーします。

operator []

char& operator [] (int index) ;

添え字演算子は，*index* 番目の位置にある文字を返します。

TParamText　　DIALOGS.H

TParamText は，***TStaticText*** からの導出クラスです。***stdio*** 関数である ***vsprintf*** を使用して書式化済みの出力が行なえるように，パラメータ付きのテキスト文字列を扱います。

データメンバ

paramCount short paramCount ;

paramCount は，***paramList*** 中のパラメータの数を示します。

関連項目 ***TParamText::paramList***

paramList void *paramList ;

paramList は汎用ポインタであり，通常，ポインタの配列(または構造体)または ***long*** 値を指します。テキスト文字列の書式化済みのパラメータとして使用されます。

メンバ関数

コンストラクタ TParamText (const TRect& bounds, const char *aText, int aParamCount) ;

TStaticText (*bounds, aText*)を呼び出し，固定文字列オブジェクトの作成と初期化を行ないます。

aText には，***printf*** の書式指定子 (%[−] [nnn]X の形式のもの)を含めることができます (書式指定子は，実行時に渡されるパラメータで置き換えられます)。 *aParamCount* で指定されるパラメータ数は，データメンバの ***paramCount*** にセットされます。

コンストラクタ TParamText (StreamableInit streamableInit) ; **protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ (組み立て関数) を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第8章も参照してください。

TP
Std class

関連項目　***TStaticText::TStaticText***, ***vsprintf*** (stdio.h)

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***, ***ipstream::readData***

dataSize　virtual ushort dataSize ( ) ;

このオブジェクトのパラメータが必要とするデータのサイズを返します。

getText　virtual void getText (char *s) ;

vsprintf (*s, text, paramList*)を呼び出すことにより，***paramList*** 内のパラメータを ***text*** 内のテキスト文字列とマージし，書式化されたテキスト文字列を作成して *s* に入れます。***text*** が空の場合には，**s* には *EOS* がセットされます。

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***, ***TStreamable***, ***ipstream***

setData　virtual void setData (void *rec) ;

paramList に ***&rec*** をセットします。

write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目　***TStreamableClass***, ***TStreamable***, ***opstream***

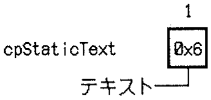
関連する関数

演算子関数で ***TParamText*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してく

ださい。

パレット

TParamText オブジェクトは，デフォルトパレット ***cpStaticText*** を使用し，標準ダイアログパレットの第6エントリに割り付けます。



TPoint OBJECTS.H

TPoint は，画面上の点を表わすクラスです。点の操作を行なう演算子がいくつかオーバーロードされます。

データメンバ

x int x;

x は，点の画面上での桁位置を表わします。。

y int y;

y は，点の画面上での行位置を表わします。

メンバ関数

operator += TPoint& operator += (const TPoint& adder) ;

x に ***adder.x*** を，*y* に ***adder.y*** を加えて *this を返します。

operator −= TPoint& operator −= (const TPoint& subber) ;

x から ***adder.x*** を，*y* から ***adder.y*** を引いて *this を返します。

フレンド

operator − friend TPoint& operator − (const TPoint& one, const TPoint& two) ;

座標 *one* から座標 *two* を引きます。*x* に (*one.x* − *two.x*)を，*y* に (*one.y* − *two.y*)をセットします。当然ながら，引数 *one* と *two* は変更されません。 *this を返します。

operator + friend TPoint& operator+(const TPoint& one, const TPoint& two) ;

座標 *one* に座標 *two* を加えます。*x* に (*one.x* | *two.x*)を，*y* に (*one.y* | *two.y*)をセットします。当然ながら，引数 *one* と *two* は変更されません。 *this を返します。

operator == friend int operator== (const TPoint& one, const TPoint& two);

点 *one* と点 *two* が同じであれば (すなわち，*one.x* == *two.x* && *one.y* == *two.y* であれば) 真 (ゼロ以外) を返します。同じでなければ偽 (0) を返します。

operator != friend int operator != (const TPoint& one, const TPoint& two) ;

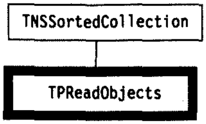
点 *one* と点 *two* が異なる場合(すなわち，*one.x* != *two.x* || *one.*

y != $two.y$ であれば）真（ゼロ以外）を返します。同じであれば偽（0）を返します。

関連する関数

この他，演算子関数で ***TPoint*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TPReadObjects　　TOBJSTRM.H



TPReadObjects（および ***TPWrittenObjects*** ）は，ストリームとの間でオブジェクトの読み書きを行なう場合に，ポインタを介して見かけ上のコピーを行なうという難題に対処します。このクラスでは，現在のオブジェクトストリームから読み出されたすべてのオブジェクトをデータベース化します。このデータベースは，***ipstream*** がストリームからポインタを読み出すときに，そのオブジェクトを示すアドレスが他にないかどうか調べるために使用されます。その結果，*ptr1* と *ptr2* が同じストリーム入出力可能なオブジェクトを指している場合に，両方のポインタを ***opstream*** に書き込もうとした場合，そのオブジェクトのコピーが 1 つだけセーブされます。逆にそのストリームから読み出しを行なう場合には，**ptr1* のコピーだけが作成され，*ptr1* と *ptr2* の両方のポインタがそのオブジェクトを指すようにされます。

メンバ関数

コンストラクタ　　TPReadObjects ( ) ;　　**private**

この private のコンストラクタは，基本クラス ***TNSSortedCollection*** のコンストラクタを呼び出し，ストリーム可能でないコレクションを作成します。メンバ関数およびフレンドからしかアクセスできません。

関連項目　***TNSSortedCollection::TNSortedCollection***

デストラクタ　~TPReadObjects ( ) ;　private

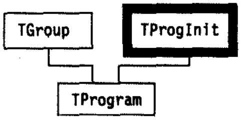
コレクションの ***limit*** を 0にします。コレクションの消去は行ないません（ ***shouldDelete*** データメンバが *False* にされているため ）。

関連項目　***TNSCollection::~TNSCollection***, ***TNSCollection::shouldDelete***

フレンド

クラス ***ipstream*** は，***TPReadObjects*** のフレンドです。したがってそのメンバ関数は，すべて ***TPReadObjects*** のプライベートメンバにアクセスすることができます。

TProgInit　APP.H



TProgInit は，***TProgram*** の仮想基本クラスです。***TProgram*** コンストラクタは，アプリケーションのステータスライン，メニューバー，およびデスクトップを作成する 3個の初期化関数のアドレスを引数にして，ベースである ***TProgInit*** のコンストラクタを呼び出してきます。***TProgram*** および ***TApplication*** のコンストラクタの項を参照してください。

メンバ関数

コンストラクタ

```
TProgInit ( TStatusLine  *( *cStatusLine)( TRect r ),
            TMenuBar  *( *cMenuBar)( TRect r ),
            TDeskTop  *( *cDeskTop )( TRect r ) );
```

TProgram のコンストラクタの説明を参照してください。

createDeskTop

```
TDeskTop  *( *createDeskTop)( TRect r );        protected
```

指定されたサイズのデスクトップを作成します。

関連項目　***TApplication::TApplication***

createMenuBar

```
TMenuBar  *( *createMenuBar)( TRect r );        protected
```

指定されたサイズのメニューバーを作成します。

関連項目　***TApplication::TApplication***

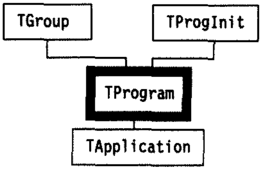
createStatusLine

```
TStatusLine  *( *createStatusLine)( TRect r );        protected
```

指定されたサイズのステータスラインを作成します。

関連項目　***TApplication::TApplication***

TProgram　　APP.H



TProgram は，Turbo Vision のすべての標準的なアプリケーションの基本となるテンプレート（雛型）です。標準的なアプリケーションプログラムは，すべて ***TProgram*** または ***TProgram*** の直接の子クラスである ***TApplication*** から導出しなければなりませ

ん。***TApplication*** と ***TProgram*** とは，デフォルトのコンストラクタとデストラクタが違うだけです。この2つのクラスは，標準的でないアプリケーションを作成したい場合に，柔軟性をもたせることができるようにするためのものです。Turbo Vision を利用する場合，たいていのアプリケーションは ***TApplication*** から導出することになります。

TProgram は，それぞれのアプリケーションで ***TDeskTop***，***TStatusLine***，および ***TMenuBar*** の各オブジェクトを持たせる必要があるため ***TGroup*** を継承します。

データメンバ

application　static TProgram　*application ;

TProgInit コンストラクタによって this にセットされた現在のアプリケーションへのポインタ。

関連項目　***TProgInit*** クラス

appPalette　static int appPalette ;

initScreen がこのアプリケーション用としてセットしたデフォルトパレットを指します。***TProgram::getPalette*** が，***appPalette*** に対応する ***TPalette*** オブジェクトを返します。

関連項目　以下のパレットの項，***TProgram::getPalette***

deskTop　static TDeskTop　*deskTop ;

現在のデスクトップオブジェクトへのポインタ。***TProgram*** コンストラクタが ***createDeskTop*** を呼び出すことによってセットします。作成されたデスクトップは，***TProgram*** グループに挿入されます。

関連項目　***TProgInit::createDeskTop***，***TProgram::initDeskTop***

menuBar　static TMenuBar　*menuBar ;

現在のメニューバーオブジェクトへのポインタ。**TProgram** コンストラクタが，***createMenuBar*** を呼び出すことによってセットします。作成されたメニューバーは，**TProgram** グループに挿入されます。

関連項目　***TProgInit::createMenuBar***，***TProgram::initMenuBar***

statusLine　static TStatusLine *statusLine ;

現在のステータスラインオブジェクトへのポインタ。**TProgram** コンストラクタが，***createStatusLine*** を呼び出すことによってセットします。作成されたステータスラインは，**TProgram** グループに挿入されます。

関連項目　***TProgInit::createStatusLine***，***TProgram::initStatusLine***

メンバ関数

コンストラクタ　TProgram ( ) ;

TProgram コンストラクタは，以下のように初期化関数のアドレスを引数にして，ベースである ***TProgInit*** のコンストラクタを呼び出します。

```
TProgram::TProgram() :
    TProgInit( &TProgram::initStatusLine, &TProgram::initMenuBar,
               &TProgram::initDeskTop
             ),
    ...
```

TProgInit コンストラクタは，以下のようにして，ステータスライン，メニューバー，デスクトップを作成します。

TP
Std class

```
TProgInit::TProgInit( TStatusLine *(*cStatusLine)( TRect ), TMenuBar
                      *(*cMenuBar)( TRect ), TDeskTop *(*cDeskTop )(
                      TRect )
                     ) :
    createStatusLine( cStatusLine ),
    createMenuBar( cMenuBar ),
    createDeskTop( cDeskTop )
    ...
```

これらの呼び出しに成功すると，それら3つのオブジェクトを ***TProgram*** グループに挿入します。そして静的なポインタである ***statusLine***, ***menuBar***, ***deskTop*** の各データメンバおよび ***application*** に，それらの新しいオブジェクトへのポインタを格納します。次に ***TGroup*** コンストラクタを呼び出し，画面全体を表わすビューを作成します。さらにビデオバッファとデフォルトパレットを初期化し，状態フラグを以下のように設定します。

```
state = sfVisible | sfSelected | sfFocused | sfModal | sfExposed;
```

関連項目　***TProgInit***, ***TGroup::TGroup***

デストラクタ　virtual ~TProgram()

このクラスにリンクされている ***deskTop***, ***menuBar***, および ***statusLine*** の各オブジェクトを消去し，静的データメンバである ***application*** を0にします。

関連項目　***~TGroup***

getEvent　virtual getEvent (TEvent& event) ;

TView::getEvent のデフォルト処理は，単にそのオーナーの ***getEvent*** を呼び出すだけになっています。すべてのビューの最終的なオーナーは，***TProgram***（または ***TApplication***）オブジェクトなので，最終的にすべての ***getEvent*** 呼び出しは，***TProgram::getEvent*** に到達することになります（その途中のどこかのビューで ***getEvent*** がオーバーライドされていないかぎり）。

TProgram::getEvent は，まず ***TProgram::putEvent*** がペンディングイベントを（静的な ***TEvent*** メンバ ***pending*** に）生成していないかどうかを調べ，ペンディングイベントが生成されていた場合には，***getEvent*** はそのイベントを返します。ペンディングがなかっ

た場合には，***getEvent*** は ***getMouseEvent*** を呼び出します。***get-MouseEvent*** が ***evNothing*** を返してきた場合には，***get-KeyEvent*** を呼び出します。***getKeyEvent*** も ***evNothing*** を返してきた場合には，ユーザ入力が何もないということを意味するので，***getEvent*** は，***TProgram::idle*** を呼び出して，アプリケーションがユーザ入力を待つ間，「バックグラウンド」タスクを実行できるようにします。***getEvent*** は，リターンする前に，***evKeyDown*** イベントおよび ***evMouseDown*** イベントがあれば，それを ***statusLine*** に渡します。これによって ***statusLine*** は，それらのイベントがそのステータスラインにリンクしている ***evCommand*** ホットキーイベントに合致するかどうかを調べることができます。

関連項目　***TProgram::putEvent***，***getMouseEvent***，***get-KeyEvent***

getPalette　virtual TPalette& getPalette ( ) const ;

静的データメンバ ***appPalette*** をパレットインデックスとして指定されるパレット文字列を返します。***TProgram*** は，***apColor***，***apBlackWhite***，および ***apMonochrome*** の 3 つのパレットをサポートしています。***appPalette*** は，***TProgram::initScreen*** で初期化されます。

関連項目　***TProgram::initScreen***，***AppPalette***，***apXXXX*** 定数

TP
Std class

handleEvent　virtual handleEvent (TEvent& event) ;

GRPH-1～GRPH-9 のキーボードイベントを受け取ったときに，***command*** に ***cmSelectWindowNum*** を設定し，***infoInt*** に 1～9 を設定した ***evBroadcast*** イベントを発行します。***TWindow::handleEvent*** がこのブロードキャストイベントを受け取ると，指定された番号のウィンドウがあればそのウィンドウを選択します。

command が ***cmQuit*** の ***evCommand*** イベントを受け取った場合には，***endModal (cmQuit)*** を呼び出します。その結果，アプリケーションは終了します。

TProgram::handleEvent は，ほとんどの場合，アプリケーションに固有なコマンドを処理できるようにするためにオーバーライドさ

れます。

関連項目　***TGroup::handleEvent***

idle　virtual void idle ( ) ;

idle は，イベントキューが空になったときに ***TProgram::getEvent*** から呼び出され，ユーザ入力を待つ間，アプリケーションがバックグラウンドタスクを実行できるようにします。

TProgram::idle は，デフォルト処理として ***statusLine－＞update*** を呼び出し，ステータスラインが現在のヘルプコンテキストに一致するようにさせます。次に，前回の ***TProgram::idle*** 呼び出し以降にコマンドセットの変更があった場合には，***command*** に ***cmCommandSetChanged*** をセットした ***evBroadcast*** を発行し，そのコマンドセットに関係のあるビューが，自分自身を有効にしたり無効にしたりすることができるようにします。

アプリケーションで ***idle*** をオーバーライドする場合には，必ず継承した ***idle*** を呼び出すようにしてください。また ***idle*** で実行するタスクは，人がわかる程度に長い時間アプリケーションを停止しないようにしてください。そうしないと，ユーザ入力の妨げとなり，アプリケーションの応答がよくないような感じを与えてしまうことになります。

initDeskTop　static TDeskTop　*initDeskTop (TRect) ;

この関数のアドレスが，***TProgInit*** コンストラクタに渡されます。***TProgInit*** コンストラクタが，アプリケーションの ***TDeskTop*** オブジェクトを作成し，そのオブジェクトへのポインタを ***deskTop*** グローバル変数に格納します。直接 ***initDeskTop*** を呼び出さないようにしてください。***initDeskTop*** はほとんどの場合，デフォルトの空の ***TDeskTop*** のかわりにユーザが定義した ***TDeskTop*** をインスタンス化するようにオーバーライドされます。

関連項目　TProgram::TProgram，TDeskTop

initMenuBar　static TMenuBar　*initMenuBar (TRect) ;

この関数のアドレスが，***TProgInit*** コンストラクタに渡されます。

TProgInit コンストラクタが，アプリケーションの ***TMenuBar*** オブジェクトを作成し，そのオブジェクトへのポインタを静的なデータメンバ ***menuBar*** に格納します。直接 ***initMenuBar*** を呼び出さないようにしてください。ほとんどの場合 ***initMenuBar*** はデフォルトの空の ***TMenuBar*** のかわりにユーザが定義した ***TMenuBar*** をインスタンス化するようにオーバーライドされます。

関連項目　TProgram::TProgram, TMenuBar

initScreen　virtual void initScreen ( ) ;

画面モードが初期化されるか変更されたときに，***TProgram::TProgram*** および ***TProgram::setScreenMode*** から呼び出されます。このメンバ関数は，画面モードに依存する変数（影のサイズ，マーカ，およびアプリケーションパレットの変数）を更新したり再設定するためのものです。

関連項目　***TProgram::TProgram***, ***TProgram::SetScreenMode***

initStatusLine　static TStatusLine *initStatusLine (TRect) ;

この関数のアドレスが，***TProgInit*** コンストラクタに渡されます。***TProgInit*** コンストラクタが，アプリケーションの ***TStatusLine*** オブジェクトを作成し，そのオブジェクトへのポインタを静的なデータメンバ ***statusLine*** に格納します。直接 ***initStatusLine*** を呼び出さないようにしてください。ほとんどの場合 ***initStatusLine*** は，デフォルトの空の ***TStatusLine*** のかわりにユーザが定義した ***TStatusLine*** をインスタンス化するようにオーバーライドされます。

関連項目　***TProgram::TProgram***, ***TStatusLine***

outOfMemory　virtual void outOfMemory ( ) ;

lowMemory が ***True*** であることを検出したときに，***TProgram::validView*** がこの ***outOfMemory*** を呼び出します。***outOfMemory*** では，メモリ不足で処理を完了できないことをユーザに警告するようにしてください。たとえば，STDDLG ヘッダファイルに

ある ***messageBox*** ルーチンを使用し，以下のようにすればよいでしょう。

```
void TMyApp::outOfMemory
{
   messageBox("Not enough memory to complete operation.", 0, mfError +
              mfOKButton);
};
```

関連項目　***TProgram::validView***，***lowMemory***

putEvent　virtual void putEvent (TEvent& event) ;

TView::putEvent のデフォルト処理は，単にそのオーナーの ***putEvent*** を呼び出すだけになっています。すべてのビューの最終的なオーナーは，***TProgram***（または ***TApplication***）オブジェクトですので，最終的にすべての ***putEvent*** 呼び出しは，***TProgram::putEvent*** に到達することになります（その途中のどこかのビューで ***putEvent*** がオーバーライドされていないかぎり）。

TProgram::putEvent は，***event*** 構造体のコピーをグローバル変数 ***pending*** に保存します。次に ***TProgram::getEvent*** の呼び出しがあったときに，そのコピーが返します。

関連項目　TProgram::getEvent，TView::putEvent

run　virtual void run ( ) ;

execute メンバ関数（***TGroup*** から継承される）を呼び出し，***TProgram*** を実行します。

関連項目　***TGroup::execute***

setScreenMode　void setScreenMode (ushort mode) ;

画面モードを設定します。*mode* は，***smNColor***, ***smMono***, ***smEColor*** の中のいずれかです。***setScreenMode*** は，マウス表示をやめ，***TScreen::setVideoMode*** を呼び出して画面モードを変更し，画面バッファを設けます。次に，画面モードに関係のある変数をすべて初期化し，***changeBounds*** を呼び出して新しい画面矩形領域を設定し，最後にマウス表示を再開します。

関連項目　*TScreen::setVideoMode*，*smXXXX* 定数

shutDown　virtual void shutDown ( ) ;

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するために内部的に使用します。*shutDown* は，*destroy* が呼び出されたときに，関連するデータメンバの設定を適切に行なえるようにするために多くのクラスでオーバーライドされます。

関連項目　第 6 章『**信頼性の高いプログラムを開発するには**』

validView　TView　*validView (TView　*p) ;

新たにインスタンス化されたビュー　**p* が有効であるかどうかを調べ，有効である場合には *p* を，有効でない場合には 0 を返します。まず *p* が 0 である場合には，0 を返します。次に *validView* 呼び出しの時点で *lowMemory* が *True* となっている場合には，*p* で指定されたビューを消去し，*outOfMemory* を呼び出して 0 を返します。次に，p－＞Valid(0) を呼び出して *False* が返ってきたら，*p* を消去し，0 を返します。それ以外の場合は，ビューは有効とみなされ *p* を返します。

validView は，新しいビューをオーナーグループに挿入する前に，そのビューが有効であることを確認するためによく使用されます。その例として，以下の文はデスクトップに新しいウィンドウを挿入するときの，インスタンス化，有効性の確認，および挿入という，通常行なわれる一連のシーケンスを示しています（*TProgram::validView* と *TGroup::insert* は，どちらも，エラーがあったときにヌルポインタが返されたら，それを無視するようになっています）。

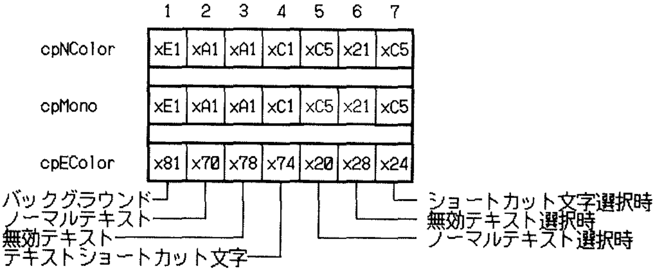
```
deskTop->insert(validView(new TMyWindow));
```

関連項目　*lowMemory*，*TProgram::outOfMemory*，*valid* メンバ関数

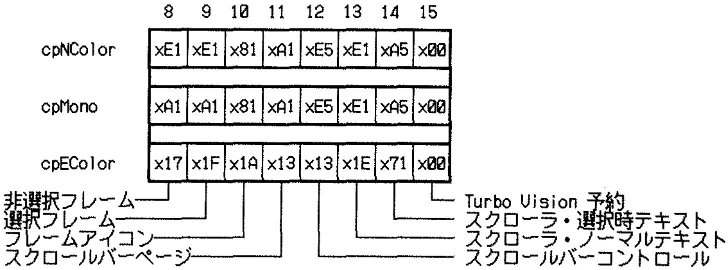
パレット

アプリケーションオブジェクトのパレットは，アプリケーション中のすべてのビューの最終的なカラー割り付けを決定します。最終的に他のすべてのパレット割り付けは，テキスト属性を決めているアプリケーションパレットのエントリを選択することに帰着します。

1番目のエントリは，***TBackground*** で背景色として使用されます。エントリ 2～7は，メニュービューとステータスラインとで使用されます。



エントリ 8～15は，ブルーウィンドウで使用されます。



エントリ16～23は，シアンウィンドウで使用されます。

	16	17	18	19	20	21	22	23
cpNColor	xA1	xA1	x81	xA1	xA1	xA1	xA5	x00
cpMono	xA1	xA1	x81	xA1	xA1	xA1	xA5	x00
cpEColor	x31	x3F	x3A	x31	x31	x3E	x21	x00

非選択フレーム
選択フレーム
フレームアイコン
スクロールバーページ
Turbo Vision 予約
スクローラ・選択時テキスト
スクローラ・ノーマルテキスト
スクロールバーコントロール

シアンウィンドウは，拡張/通常属性モードの場合は水色を基調とした色，モノクロモードでは，白の反転を基調とした色を持つウィンドウになります。

エントリ 24～31は，グレーウィンドウで使用されます。

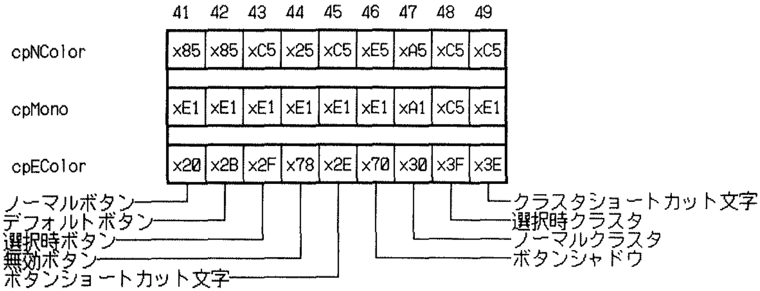
	24	25	26	27	28	29	30	31
cpNColor	xE1	xE1	x81	xE1	xE1	xE1	xE1	x00
cpMono	xA1	xA1	x81	xA1	xE1	xE1	xA5	x00
cpEColor	x70	x7F	x7A	x70	x70	x70	x7F	x00

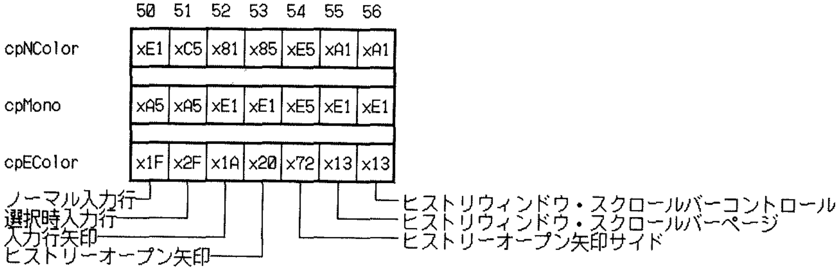
非選択フレーム
選択フレーム
フレームアイコン
スクロールバーページ
Turbo Vision 予約
スクローラ・選択時テキスト
スクローラ・ノーマルテキスト
スクロールバーコントロール

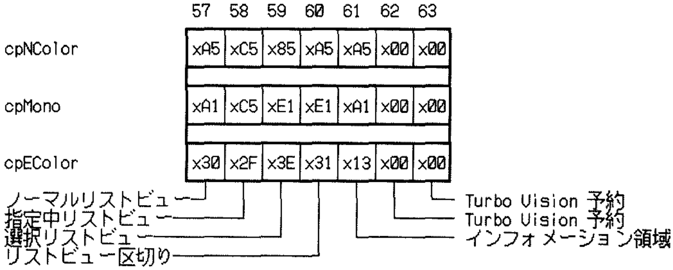
エントリ 32～63は，ダイアロボックスオブジェクトで使用されます。個々のエントリについては，***TDialog*** を参照してください。

	32	33	34	35	36	37	38	39	40
cpNColor	xE5	xE5	xE5	xA5	xA5	xE5	xE5	xE5	xC5
cpMono	xE5	xE5	xE5	xE1	xE1	xE5	xE5	xE5	xE1
cpEColor	x70	x7F	x7A	x13	x13	x70	x70	x7F	x7E

非選択フレーム
選択フレーム
フレームアイコン
スクロールバーページ
スクロールバーコントロール
ラベルショートカット文字
ハイライトラベル
ノーマルラベル
固定文字列

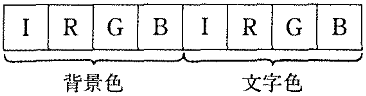






各エントリの値は，次のようなビットフィールドで指定します。

cpEColor



拡張アトリビュートを備えた機種のでみ表示できます。

I　輝度
R　赤
G　緑
B　青
すべて1のとき有効

このパレットは，ノーマルアトリビュートモードの色設定に用います。

cpNColor と cpMono

G	R	B	VL	UL	RV	BL	ST

G　緑
R　赤
B　青
VL 縦線付加
UL 下線付加
RV 反転
BL 点滅
G~BL：1のとき有効
ST 非表示　　1で表示，0で非表示

このパレットは，ノーマルアトリビュートモードの色設定に用います。

TP
Std class

TPWObj　　TOBJSTRM.H

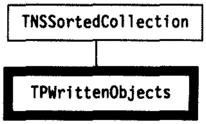
TPWObj

TPWObj は，***TPWrittenObjects*** によって内部的に使用されます。

フレンド

クラス ***TPWrittenObjects*** は，***TPWobj*** のフレンドです。したがって，***TPWrittenObjects*** の メンバ関数は，***TPWObj*** のプライベートメンバにアクセスすることができます。

TPWrittenObjects　　　　TOBJSTRM.H



TPWrittenObjects (および ***TPReadObjects***) は，ストリームとの間でオブジェクトの読み書きを行なう場合に，ポインタを介して見かけ上のコピーを行なうという難題に対処します。このクラスでは，現在のオブジェクトストリームから読み出されたすべてのオブジェクトをデータベース化します。このデータベースは，***opstream*** がストリームにポインタを書き込むときに，そのポインタが指しているオブジェクトがすでにストリームに書き込まれていないかどうかを調べるために使用されます。その結果，*ptr1* と *ptr2* が同じストリーム入出力可能なオブジェクトを指している際に，両方のポインタを ***opstream*** に書き込もうとした場合，そのオブジェクトのコピーが１つだけセーブされます。逆にそのストリームから読み出しを行なう場合には，**ptr1* のコピーだけが作成され，*ptr1* と *ptr2* の両方のポインタがそのオブジェクトを指すようにされます。

メンバ関数

コンストラクタ　　TPWrittenObjects ( ) ;　　　　private

これは，プライベートのコンストラクタです。基本である ***TNSSortedCollection*** のコンストラクタを呼び出すことによって，ストリーム入出力可能でないコレクションを作成します。メンバ関数およびフレンドからしかアクセスできません。

関連項目　***TNSSortedCollection::TNSortedCollection***

デストラクタ　~TPWrittenObjects ( ) ;　　private

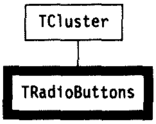
コレクションの ***limit*** を 0にします。コレクションの消去は行ないません（***shouldDelete*** データメンバが ***False*** にされているため)。

関連項目　***TNSCollection::~TNSCollection***, ***TNSCollection::shouldDelete***

フレンド

クラス ***opstream*** は，***TPWrittenObjects*** のフレンドです。したがって ***opstream*** のメンバ関数は，すべて ***TPWrittenObjects*** の プライベートメンバにアクセスすることができます。

TRadioButtons　　DIALOGS.H



TRadioButtons オブジェクトは，最大で ***maxCollectionSize*** (16,380)個のコントロールを備えることができ，常にどれか1つのコントロールボタンしか選択できないようになっているクラスタです。選択されていないボタンを選択すると，それまで選択されていたボタンは，自動的にもとの状態に戻されます。コンストラクタやデストラクタなど，ほとんどの機能は ***TCluster*** から継承されます。ラジオボタンには，普通 ***TLabel*** オブジェクトがリンクされます。

TRadioButtons は，継承した ***TCluster::value*** データメンバを「押下された」ボタンの番号として扱います。番号は，クラスタ内の 1番目のボタンが 0です。

メンバ関数

コンストラクタ　TRadioButtons (StreamableInit) ;　protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なう際に，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

draw　virtual void draw ( ) ;

ボタンを，箱で囲んだ "( )"で表示します。

mark　virtual Boolean mark (int item) ;

item が ***value*** に等しい場合，すなわち *item* 番目のボタンが現在の ***value*** データメンバ(「押下された」ボタン)に対応していれば，***True*** を返します。

関連項目　***TCluster::value***，***TCluster::mark***

movedTo　virtual void movedTo (int item) ;

value に *item* を代入します。

関連項目　***TCluster::movedTo***，***TRadioButtons::mark***

press　virtual void press (int item) ;

value に *item* を代入します。*item* 番目のボタンが押されたときに呼び出されます。

setData　virtual void setData (void *rec) ;

TCluster::setData を呼び出して ***value*** を設定し，***sel*** に ***value*** を代入します。これは，起動時には選択項目が「押下」ボタンとなっているからです。

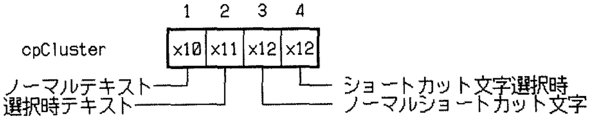
関連項目　***TCluster::setData***

関連する関数

演算子関数で ***TRadioButtons*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

TRadioButtons オブジェクトは，すべてのクラスタオブジェクトに共通のデフォルトパレットである ***cpCluster*** を使用し，ダイアログの標準パレットのエントリ 16～18に割り付けます。



TRect　　OBJECTS.H

オブジェクト ***TRect*** は，2個の ***TPoint*** オブジェクト（矩形の左

上隅と右下隅）および矩形の操作を行なうためのインラインメンバ関数からなります。2個の矩形を自然な方法で比較できるように，==と != の演算子がオーバーロードされます。***TPoint*** は，画面上での点の座標を表わす *x* および *y* をデータメンバとしています。

データメンバ

a　TPoint a;

a は，画面上の矩形領域の左上隅を表わす点です。

関連項目　***TPoint***

b　TPoint b;

b は，画面上の矩形領域の右下隅を表わす点です。

関連項目　***TPoint***

メンバ関数

コンストラクタ　TRect (int ax, int ay, int bx, int by) ;
TRect (TPoint topleft, TPoint bottomright) ;

TRect オブジェクトを作成し，*a.x = ax; a.y = ay;* というように初期化します。この方法とは別に，引数に ***TPoint*** を 2 個指定して，矩形を作成することもできます。その場合，***a*** には ***topleft*** が，***b*** には ***bottomright*** が代入されます。

コンストラクタ　TRect ( ) ;

引数を指定せずに呼び出すことによって，***new*** を使用して ***TRect*** オブジェクトを作成し，初期化を行なわないようにすることもできます。

contains　Boolean contains (const TPoint& p) const ;

呼び出している矩形領域(境界も含む)内に点 *p* がある場合に，*True*

を返します。

grow　void grow (int aDX, int aDY) ;

a.x から *aDX* を引き，*b.x* に *aDX* を加え，*a.y* から *aDY* を引き，*b.y* に *aDY* を加えることによって，呼び出している矩形領域のサイズを変更します。

intersect　void intersect (const TRect& r) ;

呼び出している矩形領域の位置とサイズを，現在の矩形領域と *r* との交わりで定義される矩形領域に変えます。

isEmpty　Boolean isEmpty ( ) ;

矩形内部に 1文字分の領域もない場合(空の場合)に *True* を返します。そうでない場合には *False* を返します。空とは，(*a.x* >= *b.x* || *a.y* >= *b.y*)のことです。

move　void move (int aDX, int aDY) ;

a.x と *b.x* に *aDX* を加え，*a.y* と *b.y* に *aDY* を加えることによって，呼び出している矩形を移動します。

operator ==　Boolean operator == (const TRect& r) const ;

r が，呼び出している矩形と同じである場合に *True* を返します。同じでない場合には *False* を返します。

operator !=　Boolean operator != (const TRect& r) const ;

r が，呼び出している矩形と同じでない場合に *True* を返します。同じである場合には *False* を返します。

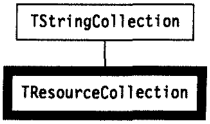
Union　void Union (const TRect& r) ;

呼び出している矩形を，それ自身と矩形 *r* との和 (union)，すなわちそれ自身と *r* の両方の矩形を包含する最小の矩形にします。

関連する関数

演算子関数で ***TRect*** に関係するものもありますが，メンバ関数とはなっていません。詳細については322ページを参照してください。

TResourceCollection　　RESOURCE.H



TResourceCollection は ***TStringCollection*** の導出クラスであり，***TStringCollection*** をソート済みのストリーム入出力可能なコレクションとします。***TResourceFile*** といっしょに使用することで，リソースのコレクションを実現することができます。リソースファイルとは，キーとなる文字列をインデックスにしてアクセスすることのできるストリームのことです。リソースの各要素は，次のように定義される ***TResourceItem*** 型のオブジェクトへのポインタです。

```
struct TResourceItem
{
   long pos;
   long size;
   char *key;
};
```

これらのフィールドは，文字列 ***key*** に対応するリソース要素のストリーム上での位置と要素のサイズを指定するものです。***TResourceCollection*** でオーバーライドを行なっているメンバ関数は，主として各要素で文字列を拡張した部分の処理を行なっています。***TResourceCollection*** は，***TResourceFile*** オブジェクトが内部的にリソースファイルのインデックスを管理するために使用されます。

データメンバ

name static const char *const name ;

ストリームマネージャが使用するクラス名。

メンバ関数

コンストラクタ TResourceCollection (short aLimit, short aDelta) ;

初期サイズ(要素数)が *aLimit* で，*aDelta* 個単位でサイズを変更することのできるリソースコレクションを作成します。

コンストラクタ TResouceCollection (StreamableInit streamableInit) ;

protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第8章も参照してください。

関連項目 ***TStringCollection::TStringCollection***

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***，***ipstream::readData***

freeItem virtual void freeItem (void *item) ;

コレクションから指定された要素を削除します。キーと要素の両方を削除します。

keyOf virtual void *keyOf (void *item) ;

指定された要素のキーを返します。定義は以下のとおりです。

```
void* TResourceCollection::keyOf( void *item )
{
   return ((TResourceItem *)item)->key;
}
```

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass*, *TStreamable*, *ipstream***

readItem void *TResourceCollection::readItem (ipstream& is) ;

コレクション内の要素ごとに呼び出されます。***TCollection*** または ***TSortedCollection*** から導出したすべてのクラスで，この種の関数をオーバーライドして，要素を正しく読み出すようにする必要があります。***TSortedCollection*** では，すでにこの関数をオーバーライドしています。

関連項目 ***TStreamableClass*, *TStreamable*, *ipstream***

write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目 ***TStreamableClass*, *TStreamable*, *opstream***

writeItem void TResourceCollection::writeItem (void *obj, opstream& os);

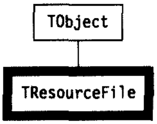
コレクション内の要素ごとに呼び出されます。***TCollection*** または ***TSortedCollection*** から導出したすべてのクラスではこの種の関数をオーバーライドして要素を正しく書き込むようにする必要があります。***TSortedCollection*** では，すでにこの関数をオーバーライドしています。

関連項目 ***TStreamableClass*, *TStreamable*, *opstream***

関連する関数

演算子関数で *TResourceCollection* に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TResourceFile　　RESOURCE.H



TResourceFile は，文字列キーをインデックスにしてアクセスすることのできるストリーム（***fpstream*** 型）を実現します。***TResourceFile::put*** を使用してオブジェクトをリソースファイルにセーブする場合，オブジェクトを識別するために文字列キーも指定します。後でそのオブジェクトを読み出す場合には，***TResourceFile::get*** 呼び出しで文字列キーを指定します。

TResourceFile オブジェクトは，リソースファイルに格納されているオブジェクトを高速かつ効率よくアクセスできるように，リソースファイル中のリソースデータの位置と位置といっしょに，そのキーを（***TResourceCollection*** クラスを使用した）ソート済み文字列コレクションにセーブします。データメンバ ***index*** には，そのリソースファイルへのインデックスとしてリンクされている ***TResourceCollection*** オブジェクトへのポインタが入ります。

ストリーム入出力についてはいつでもそうですが，リソースファイルとの間で読み書きされるオブジェクトは，すべてストリーム入出力可能なクラスでなければならず，またそのクラスを登録する必要があります（すなわち，ストリームマネージャに知らせておく必要があります。第 8 章『**ストリーム入出力可能なオブジェクト**』を参照してください）。

データメンバ

basePos　long basePos ;

ストリームのベース位置（ヘッダ情報は無視しての位置)。

関連項目　***fpstream***

index　TResourceCollection *index;

リンクしている ***TResourceCollection*** オブジェクトへのポインタ。

関連項目　***TResourceCollection***

indexPos　long indexPos ;

ベース位置相対の，ストリームの現在位置。

modified　Boolean modified ;

前回の ***flush*** 呼び出し以降に，リソースファイルに変更があった場合には *True* がセットされます。なければ *False* がセットされます。

関連項目　***TResourceFile::flush***, ***TResourceFile::put***

stream　fpstream *stream ;

このリソースファイルにリンクされているファイルストリームへのポインタ。

関連項目　***fpstream***

メンバ関数

コンストラクタ　TResourceFile(fpstream *aStream);

aStream で指定されたストリームを使用してリソースファイルを

初期化し，データメンバ ***modified*** を *False* にします。その場合，ストリームは初期化済みでなければなりません。たとえば以下のようにします。

```
TResourceFile *resFile = new TResourceFile(new fpstream("MYAPP.RES",
                                             ios::in | ios::out));
```

初期化に際して ***TResourceFile*** コンストラクタは，ストリームの現在位置にリソースファイルヘッダを探そうとします。ヘッダが見つからなかった場合には，コンストラクタはリソースファイルとリソースコレクションを新規に作成するものとみなします。通常，プログラマがヘッダ内部のことに関知する必要はありませんが，高度なプログラミングを行なう場合には，以下の情報が必要となるかもしれません。リソースファイルヘッダは，以下の構造体で定義されています。

```
struct THeader
{
   ushort signature;
   union
   {
      Count_type count;
      Info_type info;
   };
};
```

Count_type の定義は以下のとおりです。

```
struct Count_type
{
   ushort lastCount;
   ushort pageCount;
};
```

また ***Info_type*** の定義は以下のとおりです。

```
struct Info_type
{
   ushort infoType;
   long infoSize;
};
```

signature は，0x5a4d か 0x4246のいずれかです。***signature*** が 0x5a4d であれば，共用体には ***Count_type*** フィールドが使用され，

TR
Std class

signature が 0x4246であれば，***Info_type*** フィールドが使用されます。コンストラクタが，ストリームの現在位置に .EXE ファイルのサイン（***signature***）を検出した場合には，.EXE ファイルのバイナリイメージの最後のところまでそのストリームをシークしていき，そこからリソースファイルヘッダを探していきます。同様にコンストラクタは，.EXE ファイルの最後に追加されているオーバーレイファイルも飛ばしていきます。したがって，アプリケーションの .EXE ファイルの後ろに，オーバーレイファイルを追加することもリソースファイルを追加することもできます（どちらが先でもかまいません）。いずれの場合でも，***basePos*** と ***indexPos*** には，ヘッダに応じて正しい値が設定されます。

関連項目　***~TResourceFile***

デストラクタ　~TResourceFile ( ) ;

TResourceFile::flush を使用し，リソースファイルをフラッシュしてから，インデックスとストリームを消去します。

関連項目　***TResourceFile*** コンストラクタ，***TResourceFile::flush***

count　short count ( ) ;

index－＞getCount を呼び出し，リンクしている ***TResourceCollection*** に格納されているリソース要素の数を返します。

関連項目　***TResourceFile::getCount***

flush　void flush ( ) ;

前回の ***flush*** 呼び出し以降にリソースファイルに変更がなければ（すなわち ***modified*** が *False* なら），***flush*** は何もしません。変更があった場合には，ストリームの最後に更新されたインデックスをセーブし，ストリームの先頭にあるリソースヘッダを更新します。そして ***stream－＞flush*** を呼び出して ***modified*** を *False* にリセットします。

関連項目　***~TResourceFile***，***TResourceFile::modified***，***opstream::flush***

get　void *get (const char *key) ;

リンクされているリソースファイルコレクション(ポインタ ***index*** で指定されている)から指定の ***key*** を検索します。キーが見つからなかった場合には，0 を返します。キーが見つかった場合には，***get*** は，***key*** のところにある ***TResourceItem*** オブジェクトの ***pos*** フィールドで指定される位置のところまでストリームを ***seekg*** します。(***basePos*** ＋ ***pos***)の位置にあるオブジェクトを作成し，そのオブジェクトへのポインタを返します。たとえば，次のような使い方をします。

```
deskTop->insert(validView(resFile.get("editorWindow")));
```

関連項目　***TResourceCollection::at***，***TResourceFile::put***，***TApplication::validView***，***ipstream::seekg***

keyAt　const char *keyAt (short i) ;

index－＞at (i)を呼び出して，このリソースファイルの *i* 番目のキー文字列を返します。第 1 リソースのインデックスは 0，最終リソースのインデックスは ***TResourceFile::count*** －1 です。***count*** と ***keyAt*** を利用すれば，リソースファイル中の全リソースについて何らかの処理を繰り返し適用していくことができます。

関連項目　***TResourceFile::count***，***TResourceCollection::at***

TR
Std class

put　void put (TStreamable *item, const char *key) ;

item で指定されるストリーム入出力可能なオブジェクトを，*key* で指定されるキー文字列を付けてリソースファイルに追加します。そして ***modified*** を *True* にします。インデックスにすでにそのキーがある場合には，すでにあるオブジェクトを新しいオブジェクトに置き換えます。そのようなキーがなかった場合には，新しいオブジェクトはインデックスで示される適当な位置に追加されます。

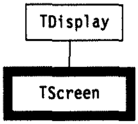
関連項目　***TResourceFile::get***，***TNSSortedCollection::search***

remove　void remove (const char *key) ;

key で示されるリソースが見つからなかった場合には，***remove*** は何もしません。そのようなリソースが見つかった場合には，***index*−>*free*** を呼び出し，そのリソースを削除します。

関連項目　***TNSSortedCollection::search***，***TNSCollection::free***

TScreen　SYSTEM.H



TScreen では，低レベルのビデオ関係の属性および機能が用意されます。このクラスを含め，SYSTEM.H にあるシステム関係のクラスについては簡単にしか紹介しません。これらのクラスは，Turbo Vision が内部的に使用するものであり，通常のアプリケーションであれば直接使用する必要はありません。なお ***TView*** は，***TDisplay*** のフレンドクラスです。

データメンバ

cursorLines　static ushort near cursorLines ;

現在のカーソルの種類が入っています。***setCrtData*** が，***getCursorType*** を呼び出して設定します。

関連項目　***TDisplay::getCursorType***，***TScreen::setCrtData***

screenHeight　static uchar near screenHeight ;

現在の画面高。***setCrtData*** が ***getRows*** を呼び出して設定します。

関連項目　***TDisplay::getRows***，***TScreen::setCrtData***

screenMode　static ushort near screenMode ;

現在のビデオモード。

関連項目　***TDisplay::getVideoMode***

screenWidth　static uchar near screenWidth ;

現在の画面幅。***setCrtData*** が ***getCols*** を呼び出して設定します。

関連項目　***TDisplay::getCols***, ***TScreen::setCrtData***

startupCursor　static ushort near startupCursor ;

カーソルの種類の初期値。***initScreen*** が ***TApplication/TProgram*** コンストラクタを介して設定します。

関連項目　***TProgram::initScreen***, ***TDisplay::getCursorType***

startupMode　static ushort near startupMode ;

ビデオモードの初期値。***initScreen*** が ***TApplication/TProgram*** コンストラクタを介して設定します。

関連項目　***TProgram::initScreen***

メンバ関数

コンストラクタ　TScreen ( ) ;

TScreen オブジェクトを作成し，***resume*** を呼び出します。***resume*** は，***startupMode*** および ***startupCursor*** を，それぞれ ***getCrtMode*** および ***getCursorType*** を使って初期化しさらに ***setCrtData*** を呼び出してその他のデータメンバの設定を行ないます。

関連項目　***TDisplay::getCrtMode***, ***TDisplay::getCursorType***, ***TScreen::setCrtData***, ***TScreen::resume***

デストラクタ　~TScreen ( ) ;

suspend を呼び出します。***suspend*** は画面モードを起動時のモードに戻し，画面をクリアしてカーソルを起動時のカーソルに戻します。

関連項目　***TDisplay::startupMode***, ***TDisplay::setCrtMode***, ***TScreen::clearScreen***, ***TDisplay::setCursorType***, ***TScreen::***

suspend

clearScreen static void clearScreen ( ) ;

現在の *screenWidth* と *screenHeight* を引数にして *TDisplay::clearScreen* を呼び出します。

関連項目 *TDisplay::clearScreen*

resume static void resume ( ) ;

TScreen コンストラクタから呼び出されます。*startupMode* および *startupCursor* を，それぞれ *getCrtMode* および *getCursorType* を使って初期化し，さらに *setCrtData* を呼び出してその他のデータメンバの設定を行ないます。

関連項目 *TDisplay::getCrtMode*, *TDisplay::getCursorType*, *TScreen::setCrtData*, *TScreen::TScreen*

setCrtData static void setCrtData ( ) ;

screenMode, *screenWidth* および *screenHeight* の各データメンバを，それぞれ *getCrtMode*, *getCols* および *getRows* を呼び出して設定します。

関連項目 *TDisplay::getCrtMode*, *TDisplay::getCols*, および *TDisplay::getRows*

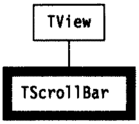
setVideoMode static void setVideoMode (ushort vmode) ;

ビデオモードを *vmode* にし，それに即して *TScreen* の他のデータメンバを再設定します。

suspend static void suspend ( ) ;

TScreen デストラクタから呼び出されます。*suspend* は画面モードを起動時のモードに戻し，画面をクリアしてカーソルを起動時のカーソルに戻します。

関連項目 *TScreen::~TScreen*



データメンバ

arStep short arStep;

arStep は，矢印領域（***sbLeftArrow***，***sbRightArrow***，***sbUpArrow***，または ***sbDownArrow***）がクリックされたとき，あるいはそれに相当するキー入力があったときに，スクロールバーのデータメンバ ***value*** に対して加減されるステップ値です。***TScrollBar*** コンストラクタが，デフォルト値として ***arStep*** に 1 を設定します。

関連項目 ***TScrollBar::setStep***，***TScrollBar::setParam***，***TScrollBar::scrollStep***

chars TScrollChars chars;

TScrollChars は，次のように定義されています。

```
typedef char TScrollChars[5];
```

chars には，スクロールバーの各部品を表示するために使用される基本的な 5 個の文字パターンが設定されます。

maxVal short maxVal;

maxVal は，データメンバ ***value*** がとり得る最大値を表わします。***TScrollBar*** コンストラクタが ***maxVal*** にデフォルト値として 0 を設定します。

関連項目 ***TScrollBar::setRange***，***TScrollBar::setParams***

minVal　short minVal;

minVal は，データメンバ ***value*** がとり得る最小値を表わします。***TScrollBar*** コンストラクタが ***minVal*** にデフォルト値として 0 を設定します。

関連項目　***TScrollBar::setRange***，***TScrollBar::setParams***

pgStep　short pgStep;

pgStep は，ページ領域のどこか（***sbPageLeft***，***sbPageRight***，***sbPageUp***，または ***sbPageDown***）でマウスクリックイベントが発生したとき，またはそれに相当するキー入力（**CTRL-←**，**CTRL-→**，**ROLLDOWN** または **ROLLUP**）が検出されたときに，スクロールバーのデータメンバ ***value*** に加減されるステップ値です。***TScrollBar*** コンストラクタが，***pgStep*** にデフォルト値として 1 を設定します。***TScrollBar::setStep*** か ***TScrollBar::setParams*** または ***TScroller::setLimit*** を使用して ***pgStep*** を変更することもできます。

関連項目　***TScrollBar::setStep***，***TScrollBar::setParams***，***TScroller::setLimit***，***TScrollBar::scrollStep***

value　short value;

データメンバ ***value*** は，スクロールバーインジケータの現在の位置を表わします。インジケータは，スクロール可能なテキストを表示する際に，現在表示中のテキストがテキスト全体の中での相対的な位置としてどこにあるかを示すために，スクロールバー領域上を（スクロールバーの向きに応じて水平方向または垂直方向に）移動するマークです。インジケータには特別な配色がなされます。

いろいろなイベントが，直接的または間接的に ***value*** の値を変更します。たとえば，スクロールバー内の特定の部位をクリックしたり，ウィンドウを拡大・縮小したり，スクローラ中のテキストを修正した場合です。同様に，***value*** を変更することによって，別のイベントを発行する必要がでてくることもあります。***TScrollBar*** コンストラクタは，***value*** にデフォルト値として 0 を設定します。

関連項目　*TScrollBar::setvalue*, *TScrollBar::setParams*, *TScrollBar::scrollDraw*, *TScroller::handlevent*, *TScrollBar::TScrollBar*

メンバ関数

コンストラクタ　TScrollBar (const TRect& bounds) ;

TView コンストラクタを呼び出し，指定した *bounds* を持つスクロールバーを作成，初期化します。***value***, ***maxVal***, ***minVal*** には 0 を設定し，***pgStep*** と ***arStep*** には 1 を設定します。スクロールバーの各部品の形状を，***TScrollChars*** に指定してあるデフォルト値にします。

bounds の指定で ***size.x*** = 1 となった場合には垂直スクロールバーが作られることになり，それ以外の場合には水平スクロールバーが作られることになります。垂直スクロールバーの ***growMode*** データメンバには ***gfGrowLoX*** | ***gfGrowHiX*** | ***gfGrowHiY*** を設定し，水平スクロールバーの ***growMode*** データメンバには，***gfGrowLoY*** | ***gfGrowHiX*** | ***gfGrowHiY*** を設定します。

コンストラクタ　TScrollBar (StreamableInit streamableInit) ;　**protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

build　static TStreamable *build () ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　*TStreamableClass*, *ipstream::readData*

draw void draw ( ) ;

現在の ***bounds***, ***value***, および パレット を使用してスクロールバーを表示します。

関連項目 ***TScrollBar::scrollDraw***, ***TScrollBar::value***

getPalette virtual const char *getPalette ( ) const ;

スクロールバーのデフォルトのパレット文字列 ***cpScrollBar***, すなわち "¥x04¥x05¥x05¥" を返します。

handleEvent virtual void handleEvent (TEvent& event) ;

TView::handleEvent を呼び出してスクロールバーイベントを処理した後, *event.what* を調べます。マウスイベントは, スクロールバーのオーナーにブロードキャストされます (***message*** 関数を参照)。オーナーは, スクロールバーの変化(たとえば, テキストのスクロール)に対応した処理を行ないます。また, ***TScrollBar::handleEvent*** は, スクロールバーの部品の中で, どれがマウスクリック (またはそれに相当するキー入力) を受け取ったのかを判断します。現在の ***arStep*** または ***pgStep*** の値にしたがってデータメンバ ***value*** を再設定し, スクロールバーインジケータの再表示を行ないます。

関連項目 ***TView::handleEvent***

read virtual void *read (ipstream& is) ;

入力ストリーム ***is*** からの読み出しを行ないます。

関連項目 ***TStreamableClass***, ***TStreamable***, ***ipstream*** クラス

scrollDraw virtual void scrollDraw ( ) ;

scrollDraw は, データメンバ ***value*** が変更されたときに呼び出されます。この仮想メンバ関数はデフォルト処理として, 以下のようにして, そのスクロールバーのオーナーに ***cmScrollBarChanged*** メッセージを発行します。

```
message(owner, evBroadcast, cmScrollBarChanged, this);
```

関連項目　***TScrollBar::value***, ***message***

scrollStep　virtual short scrollStep (short part);

scrollStep はデフォルト処理として，*part* で指定されたスクロールバーの部品および ***arStep***, ***pgStep*** の現在値を基にして正または負のステップ値を返します。引数 ***part*** には，第16章に示してあるスクロールバー部品定数 ***sbXXXX*** の中のどれかを指定しなければなりません。

関連項目　***TScrollBar::setStep***, ***TScrollBar::setParams***

setParams　void setParams (short aValue, short aMin, short aMax, short aPgStep, short aArStep);

setParams は ***value***, ***minVal***, ***maxVal***, ***pgStep*** および ***arStep*** の各データメンバに，引数で指定された値を設定します。引数間に矛盾がある場合には値の調整を行ないます。たとえば ***minVal*** には，***maxVal*** よりも大きな値を設定することはできません。したがって *aMax* ＜ *aMin* である場合には，***maxVal*** に *aMin* を設定します。また ***value*** は，［***minVal,maxVal***］の範囲内に納まっていなければなりません。したがって *aValue* ＜ *aMin* である場合には，***value*** に *aMin* を設定し，*aValue* ＞ *aMax* である場合には，***value*** に *aMax* を設定します。そして ***drawView*** を呼び出し，スクロールバーの再表示を行ないます。***value*** が変更された場合には ***scrollDraw*** も呼び出します。

関連項目　***TView::drawView***, ***TScrollBar::scrollDraw***, ***TScrollBar::setRange***, ***TScrollBar::setValue***

setRange　void setRange (short aMin, short aMax);

setRange は，引数で指定された *aMin* と *aMax* を ***minVal*** と ***maxVal*** に設定することで，データメンバ ***value*** の有効範囲を決めます。***setRange*** は ***setParams*** を呼び出します。したがって，この変更によってスクロールバーの再表示が必要になった場合には，***drawView*** および ***scrollDraw*** が呼び出されることになります。

関連項目　***TScrollBar::setParams***

setStep　void setStep (short aPgStep, short aArstep) ;

setStep は，引数に指定された *aPgStep* および *aArStep* を，データメンバ ***pgStep*** および ***arStep*** に設定します。***setParams*** のその他の引数に現在の設定値を指定して ***setParams*** の呼び出しを行ないます。

関連項目　***TScrollBar::setParams***，***TScrollBar::scrollStep***

setValue　void setValue (short aValue) ;

setValue は，データメンバ ***value*** に *aValue* を設定します。***setParams*** のその他の引数に現在の設定値を指定して ***setParams*** の呼び出しを行ないます。***value*** の値に変更がある場合には，***drawView*** および ***scrollDraw*** が呼び出されることになります。

関連項目　***TScrollBar::setParams***，***TView::drawView***，***TScrollBar::scrollDraw***，***TScroller::scrollTo***

write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***opstream***

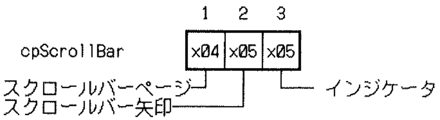
関連する関数

演算子関数で ***TScrollBar*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

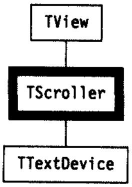
パレット

スクロールバーオブジェクトは，デフォルトのパレット ***cpScrollBar*** を使用して，標準アプリケーションパレットのエントリ 4〜5

に割り付けます。



TScroller VIEWS.H



データメンバ

delta TPoint delta ;

delta は，スクロールされる仮想的なビューに対するそのスクローラの相対的な位置を表わす ***x*** (水平) 座標，***y*** (垂直) 座標を格納します。***value*** データメンバを変更するスクロールバーイベントを受け取ったときなどに，これらの座標の中の片方または両方を変更することによって，自動スクロールを行なうことができます。一方，ユーザがスクロール操作を行なった場合でも ***delta*** は変更され，スクロールバーのデータメンバ ***value*** を変更させ，スクロールバーのインジケータを更新させることになります。

関連項目 ***TScroller::scrollDraw***，***TScroller::scrollTo***

drawFlag Boolean drawFlag ;

スクローラの再表示が必要な場合に *True* にされます。

関連項目 ***TView::drawView***，***TScroller::drawLock***，***TScrol-***

ler::checkDraw

drawLock uchar drawlock ;

スクローラの再表示を制御するために使用されるセマフォ。

関連項目 ***TView::drawView***, ***TScroller::drawFlag***, ***TScroller::checkDraw***

hScrollBar TScrollBar *hScrollBar ;

hScrollBar は，このスクローラにリンクされている水平スクロールバーオブジェクトへのポインタです。水平スクロールバーがない場合には ***hScrollBar*** は 0 です。

limit TPoint limit ;

limit.x, *limit.y* は，***delta.x***, ***delta.y*** のとり得る最大値です。

関連項目 ***TScroller::delta***

vScrollBar TScrollBar *vScrollBar ;

vScrollBar は，このスクローラにリンクされている垂直スクロールバーオブジェクトへのポインタです。垂直スクロールバーがない場合には ***vScrollBar*** は 0 です。

メンバ関数

コンストラクタ TScroller (const TRect& bounds, TScrollBar *aHScrollBar, TScrollBar *aVScrollBar) ;

指定のサイズの，スクロールバー付きの ***TScroller*** オブジェクトを作成，初期化します。***TView*** コンストラクタを呼び出してビューのサイズを設定します。***options*** には ***ofSelectable*** を，***eventMask*** には ***evBroadcast*** を設定します。水平スクロールバーを持たせたくない場合には *aHScrollBar* を 0 にし，垂直スクロールバーを持たせたくない場合には *aVScrollBar* を 0 にします。

コンストラクタ TScroller(StreamableInit streamableInit) ; **protected**

TS
Std class

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TView::TView***，***TView::options***，***TView::eventMask***

build　static TStreamable *build ( )；

ストリーム読み出しを行なう際，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

changeBounds　virtual void changeBounds (const TRect& bounds)

setbounds を呼び出してスクローラのサイズを変更します。必要なら ***drawView*** および ***setLimit*** を呼び出し，スクローラおよびスクロールバーの再表示を行ないます。

関連項目　***TView::setbounds***，***TView::drawView***，***TScroller::setLimit***

checkDraw　void checkDraw ( )；

drawLock がゼロで，かつ ***drawFlag*** が *True* なら ***drawFlag*** を *False* にして ***drawView*** を呼び出します。***drawLock*** がゼロ以外であるか ***drawFlag*** が *False* なら ***checkDraw*** は何も行ないません。***scrollTo*** および ***setLimit*** が，それぞれ ***checkDraw*** を呼び出し，必要な場合にのみ ***drawView*** を呼び出すようにします。

getPalette　virtual TPalette& getPalette ( )　const;

スクローラのデフォルトのパレット文字列 ***cpScroller***，すなわち "¥x06¥x07" を返します。

handleEvent virtual void handleEvent (TEvent& event) ;

TView::handleEvent を呼び出すことによって，ほとんどのイベントを処理します。***hScrollBar*** か ***vScrollBar*** のどちらかが発行する ***cmScrollBarChanged*** などのブロードキャストイベントを受け取ると，***TScroller::scrollDraw*** を呼び出します。

関連項目 ***TView::handleEvent***, ***TScroller::scrollDraw***

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***, ***TStreamable***, ***ipstream***

scrollDraw virtual void scrollDraw ( ) ;

delta がスクロールバーの現在位置と対応しているかどうかをチェックします。対応していない場合には ***delta*** に正しい値を設定し，***drawView*** を呼び出してスクローラの再表示を行ないます。

関連項目 ***TView::drawView***, ***TScroller::delta***, ***TScroller::hScrollBar***, ***TScroller::vScrollBar***

scrollTo void scrollTo (short x, short y) ;

hScrollBar->setValue(x) および ***vScrollBar->setValue(y)*** を呼び出してスクロールバーに （*x*, *y*） を設定し，必要なら ***drawView*** を呼び出してビューの再表示を行ないます。

関連項目 ***TView::drawView***, ***TScroller::setValue***, ***TScroller::checkDraw***

setLimit void setLimit (short x, short y) ;

データメンバ ***limit*** の設定を行ない，必要ならスクロールバーおよびスクローラの再表示を行ないます。

関連項目 ***TScroller::limit***, ***TScroller::hScrollBar***, ***TScroller::vScrollBar***, ***TScrollBar::setParams***, ***TScroller::checkDraw***

setState　virtual void setState (ushort aState, Boolean enable) ;

スクローラの状態が変更したときに呼び出されます。***TView::set-State*** を呼び出し，*aState* で状態フラグをセットまたはクリアします。更新した状態が ***sfSelected*** かつ ***sfActive*** となった場合には，***setState*** はスクロールバーの表示を行ないます。それ以外の場合には，スクロールバーの表示を行なわないようにします。

関連項目　***TView::setState***

shutDown　virtual void shutDown ( ) ;

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するために内部的に使用します。***shutDown*** は，***destroy*** が呼び出されたときに，関連するデータメンバの設定を適切に行なえるようにするために多くのクラスでオーバーライドされます。

関連項目　第6章**『信頼性の高いプログラムを開発するには』**

write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目　***TStreamableClass***, ***TStreamable***, ***opstream***

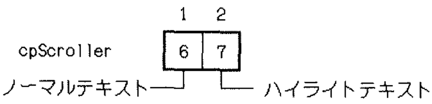
関連する関数

演算子関数で ***TScroller*** に関係するものもありますが，メンバ関数とはなっていません。詳細については322ページを参照してください。

パレット

スクローラオブジェクトはデフォルトパレット ***cpScroller*** を使用し，標準アプリケーションパレットのエントリ 6〜7 に割り付け

ます。



TSItem　　DIALOGS.H

TSItem は，文字列を要素とする単方向のリンクリストを構成するためのビューではない単純なクラスです。これは，文字列コレクションほどの機能を使うまでもないような場合（その例が ***TCluster***）に重宝するクラスです。

データメンバ

next　TSItem *next;

リンクリスト中の次の ***TSItem*** オブジェクトへのポインタ。

value　const char *value;

この ***TSItem*** オブジェクトの文字列。

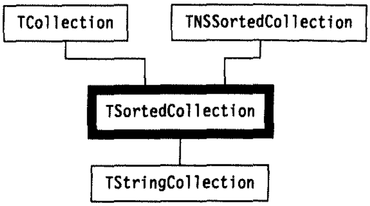
メンバ関数

コンストラクタ　TSItem (const char *aValue, ***TSItem*** *aNext);

指定した値を持つ TSItem オブジェクトを作成します。

デストラクタ　~TSItem ( );

delete value を呼び出して ***TSItem*** オブジェクトを消去します。



TSortedCollection は ***TCollection*** と ***TNSSortedCollection*** の両方を継承し，キーを使ってソートされる（キーの重複も許される）ストリーム入出力可能なコレクションを実現するために特殊化された抽象クラスです。***TSortedCollection*** を実体化することはできません。標準の導出クラスまたはユーザ定義の導出クラスを作成するときの基本としてのみ存在します。

ソートには，純粋仮想メンバ関数 ***compare*** を使用します。この関数 ***compare*** は，導出クラスでオーバーライドし，それぞれの用途に合わせて要素をソートする機能を持たせる必要があります。新しい要素を追加するときには，自動的に ***compare*** が指定する順序にしたがって挿入を行ないます。要素の配置には，***TNSSortedCollection*** から継承した ***search*** 関数を使用することができます。***compare*** で使用されるポインタを返してくる関数 ***indexOf***（これも ***TNSSortedCollection*** から継承される）も仮想となっており，***compare*** がそれ以外の情報を必要とする場合のためにオーバーライドすることができるようになっています。

ストリーム入出力可能なソート済みコレクションを作成したい場合には，***TSortedCollection*** を使用する必要があります。ストリーム入出力可能かどうかという点を除けば，***TSortedCollection*** と ***TNSSortedCollection*** の機能は同じです。

メンバ関数

コンストラクタ　TSortedCollection (ccIndex aLimit, ccIndex aDelta)；

TCollection コンストラクタを呼び出し，***count***，***items***，および ***limit*** に 0 をセットします。さらに ***setLimit***(*aLimit*) を呼び出してコレクションのサイズを *aLimit* とし，***delta*** に *aDelta* をセットします。なお，***ccIndex*** は int として typedef 宣言されています。***duplicates*** には *False* をセットします。キーを重複させてもよい場合には，***duplicates*** に *True* をセットする必要があります。

コンストラクタ TTSortedCollection (StreamableInit streamableInit) ;

protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目 ***TCollection::TCollection***，***TCollection*** データメンバ

compare virtual int compare (void *key1, void *key2) = 0 ; private

compare は，純粋仮想関数であり，導出クラスでは必ずオーバーライドする（あるいは純粋仮想として再定義する）必要があります。

関連項目 ***TNSSortedCollection::compare***

TS
Std class

read void read (ipstream& is) ; protected

入力ストリーム *is* からソート済みコレクションを読み出します。

関連項目 ***TStreamableClass***，***TStreamable***，***ipstream***

readItem void *TSortedCollection::readItem (ipstream& is) ;

コレクション内の要素ごとに呼び出されます。***TCollection*** または ***TSortedCollection*** から導出したすべてのクラスで，この種の関数をオーバーライドして要素を正しく読み出すようにする必要があります。***TSortedCollection*** では，すでにこの関数をオーバーライ

ドしています。

関連項目 ***TStreamableClass*, *TStreamable*, *ipstream***

write void write (opstream& os) ; protected

リンクしている ***TSortedCollection*** のオブジェクトを出力ストリーム *os* に書き込みます。

関連項目 ***TStreamableClass*, *TStreamable*, *opstream*** クラス

writeItem void TSortedCollection::writeItem (void *obj, opstream& os) ;

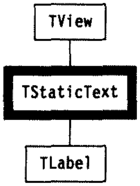
コレクション内の要素ごとに呼び出されます。***TCollection*** または ***TSortedCollection*** から導出したすべてのクラスでは，この種の関数をオーバーライドして要素を正しく書き込むようにする必要があります。***TSortedCollection*** では，すでにこの関数をオーバーライドしています。

関連項目 ***TStreamableClass*, *TStreamable*, *opstream***

関連する関数

演算子関数で ***TSortedCollection*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TStaticText DIALOGS.H



TStaticText オブジェクトは，ビューとなり得るオブジェクト

の中で，もっとも単純なオブジェクトです。このオブジェクトは固定テキストを格納し，送られてきたイベントはすべて無視します。一般に，メッセージや受動的なラベルとして使用されます。***TLabel*** オブジェクトなどの ***TStaticText*** の下位オブジェクトは，通常もっと能動的な役割を果たします。

データメンバ

text　const char *text;　**protected**

ビューに表示されるテキスト文字列（定数）へのポインタ。

メンバ関数

コンストラクタ　TStaticText (const TRect& bounds, const char *aText);

TView (*bounds*) を呼び出して指定されたサイズの ***TStaticText*** オブジェクトを作成した後，***text*** に ***newStr*** (*aText*) を設定します。

コンストラクタ　TStaticText (StreamableInit streamableInit);　**protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，***vtable*** ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TView::TView***，***newStr***

デストラクタ　~StaticText ( );

テキスト文字列を廃棄した後，~***TView*** を呼び出してオブジェクトを消去します。

build　static TStreamable *build ( );

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

draw　virtual void draw ( ) ;

ビュー内にテキスト文字列を表示します。必要ならワードラップを行ないます。テキスト中の '¥n' は改行を意味します。文字列中の行が 0x03で始まっている場合には，その行はビューの中央に配置されます。

getPalette　virtual TPalette& getPalette ( ) const ;

デフォルトのパレット文字列 ***cpStaticText***，すなわち "¥x06" を返します。

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream***

write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***opstream***

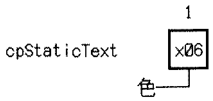
関連する関数

演算子関数で ***TStaticText*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

固定テキストオブジェクトはデフォルトパレット ***cpStaticText***

を使用し，標準ダイアログパレットのエントリ 6 に割り付けます。



TStatusDef MENUS.H

TStatusDef

TStatusDef オブジェクトは，ビューではなく，***TStatusLine*** ビューが場面に応じたステータスラインを表示する際に使用するステータスラインの定義です。データメンバ ***next*** は，ステータスラインのリストにおける次の ***TStatusDef*** オブジェクトへのポインタです（このオブジェクトが最後なら 0 です）。***min*** と ***max*** は，そのステータスラインに対応するヘルプコンテキストの範囲を定義します。***items*** は，そのステータスラインを構成するステータス項目を要素とするリストへのポインタです。

TStatusLine オブジェクトは，***TStatusDef*** オブジェクトをつないでいるリンクリストへのポインタ（***defs***）を持っています。***TStatusLine*** は，いつでも現在のヘルプコンテキスト値が ***min*** と ***max*** の範囲内にあるステータス項目のうち最初のものを表示します。***TStatusLine::update*** は定期的にステータスラインビューを更新できるように，***TProgram::idle*** から呼び出すことができるようになっています。

データメンバ

items　TStatusItem *items;

items は，ステータスラインを構成するステータス項目を要素とするリストへのポインタです。0 の場合には，ステータス項目が何もないことを意味します。

関連項目　***TStatusLine*** クラス，***TStatusItem*** クラス

min, max　ushort min, max;

このステータスラインの定義がリンクされているヘルプコンテキストの最大値と最小値。***TStatusLine*** は，いつでも現在のヘルプコンテキスト値が ***min*** と ***max*** の範囲内にあるステータス項目のうち最初のものを表示します。

関連項目　***TStatusLine::draw***

next　TStatusDef *next;

next は，0 以外の値をとる場合，ステータス定義のリスト中での次の ***TStatusDef*** オブジェクトへのポインタです。0 なら，この ***TStatusDef*** オブジェクトがリスト中の最後のオブジェクトであることを意味します。

メンバ関数

コンストラクタ

```
TStatusDef ( ushort aMin, ushort aMax,
             TStatusItem *someItems,
             TStatusDef *aNext );
```

指定した値を持つ ***TStatusDef*** オブジェクトを作成します。

TStatusItem　　MENUS.H

TStatusItem オブジェクトはビューではなく，***TStatusLine*** ビューにリンクされたリンクリストの要素（ステータス項目）を表わします。***TStatusLine*** ビューはステータスラインを表示したり，ホットキーや項目上でのクリックなどのステータスラインイベントを処理したりすることができます。ステータス項目は，表示させる

ことも表示させないようにすることもできます。表示させない場合，項目はホットキーの定義だけに使用されます。***next*** は，リスト中の次の ***TStatusItem*** を指すポインタです。0 の場合には，そのオブジェクトがリストの中の最後のオブジェクトであることを示します。***text*** は，項目の見出しとなる文字列です（"~Grph-X~ Exit" など)。***text*** が 0 の場合は，ステータス項目は表示されません。***keyCode*** には，その項目にリンクされたホットキーのスキャンコードが入ります。0 の場合には，その項目にホットキーがリンクされていないことを意味します。***command*** には，ホットキーやマウスクリックによってその項目が選択されたときに発行されるコマンドイベントが入ります。

TStatusItem には 2 つの目的があります。1 つは，ステータスラインの視覚的な形態を決定することであり，もう 1 つは，キーコードをコマンドに対応付けることによりホットキーを定義することです。

TProgram::getEvent は，すべての ***evKeyDown*** イベントについて ***TStatusLine::handleEvent*** を呼び出し，現在のステータスラインにキーコードの一致する項目がないか検索します。一致する項目があった場合には，***evKeyDown*** イベントをその項目オブジェクトの ***command*** データメンバで指定されている ***evCommand*** イベントに変換します。

データメンバ

command ushort command ;

このステータス項目にリンクされているコマンドの値。

keyCode ushort keyCode ;

リンクされているホットキーのスキャンコード。

next TStatusItem *next ;

next は，0 以外の値をとる場合，ステータスラインにリンクされているリンクリスト中での次の ***TStatusItem*** オブジェクトへのポ

インタです。0 なら，この オブジェクトがリスト中の最後の項目であることを意味します。

text const char *text;

このステータス項目に表示されるテキスト文字列。0 なら見出しは表示されません。つまり，そのステータス項目は，***keyCode*** メンバでホットキーを定義するためだけのものであることになります。

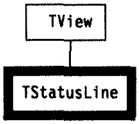
関連項目 ***TStatusLine*** クラス

メンバ関数

コンストラクタ

```
TStatusItem( const char *aText, ushort key, ushort cmd,
             TStatusItem *aNext = 0 );
```

指定した値を持つ ***TStatusItem*** オブジェクトを作成します。



TStatusLine オブジェクトは，通常画面の最下段に表示される特殊化されたビューです。ステータスラインには，通常使用することのできる一連のホットキー，使用可能なメモリサイズ，日時，現在の編集モード，ユーザへのヒントが表示されます。表示される各項目は，***TApplication*** が ***initStatusLine*** を呼び出してリンクリストにして用意します。その中の1つが，現在フォーカスのあるビューのヘルプコンテキストにしたがって表示されます。メニューバー，デスクトップと同様，ステータスラインも，通常，***TApplication*** グループによって所有されます。

ステータスラインの各項目は ***TStatusItem*** オブジェクトです。***TStatusItem*** オブジェクトには，ステータスラインに表示されるテキスト文字列，ホットキーに対応するキーコード（通常，ファンクションキーまたは ***GRPH***＋キー）および項目として表示されているテキストをマウスクリックしたり，ホットキーの入力があったときに発行されるコマンドをデータメンバとして用意しています。

各ステータスラインオブジェクトは，ステータスライン定義（***defs***：***TStatusDef*** クラスのオブジェクト）のリンクリストを備えています。***defs*** は，ヘルプコンテキストの範囲と，現在のヘルプコンテキストがその範囲内にあるときに表示されるステータス項目のリストを定義します。この他にも，現在のヘルプコンテキストにしたがってヒントや既定義の文字列を表示させることもできます。

データメンバ

defs TStatusDef *defs;

TStatusDef オブジェクトからなるリンクリストの中，現在のリンクリストへのポインタ。現在のリンクリストは，現在のヘルプコンテキストによって決まります。

関連項目　***TStatusDef***, ***TStatusLine::update***, ***TStatusLine::hint***

items　TStatusItem *items;

TStatusItem レコードからなるリンクリストの中で，現在のリンクリストへのポインタ。

関連項目　***TStatusItem***

メンバ関数

コンストラクタ　TStatusLine (const TRect& bounds, TStatusDef& aDefs) ;

TView (*bounds*) を呼び出して指定した *bounds* を持つ ***TStatusLine*** オブジェクトを作成します。***options*** の ***ofPreProcess*** ビットを立て，***eventMask*** には ***evBroadcast*** を入れ，***GrowMode*** は ***gfGrowLoY*** | ***gfGrowHiX*** | ***gfGrowHiY*** とします。データメンバ ***defs*** には *aDefs* を設定します。*aDefs* が 0 の場合は，***items*** に 0 を入れ，0 でない場合には ***items*** に *aDefs->items* を入れます。

コンストラクタ　TStatusLine(StreamableInit streamableInit);　　**protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TView::TView***

デストラクタ　~TStatusLine ( ) ;

その ***TStatusLine*** オブジェクトの ***items*** と ***defs*** をすべて廃棄した後，***~TView*** を呼び出します。

関連項目　***~TView***

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

draw　virtual void draw ( ) ;

各ステータス項目が持っているテキスト文字列を書いていくことによりステータスラインを表示します。さらに区切りを示す縦線に続けて，現在のヘルプコンテキストでヒントが定義されていればそれを表示します。

draw は各項目の状態に応じて，***cpNormal***，***cpSelect***，***cpNormDisabled*** または ***cpSelDisabled*** の中のいずれかのパレットを使用します。

関連項目　***TStatusLine::hint***

getPalette　virtual const char *getPalette ( )　const ;

デフォルトのパレット文字列 ***cpStatusLine***，すなわち "¥x02¥x03¥x04¥x05¥x06¥x07" を返します。

handleEvent　virtual void handleEvent (TEvent& event) ;

ステータスラインに送られてきたイベントの処理を行ないます。まず ***TView::handleEvent*** を呼び出した後，以下の3種類の特別なイベントの処理を行ないます。ステータス項目のいずれかが占めている矩形領域内で行なわれたマウスクリックに対しては，そのステータス項目の ***command*** を *event.what* に設定したコマンドイベントを発行します。キーイベントについては，各項目の ***keyCode*** データメンバに一致するものがないかチェックします。一致するキーイベントがあった場合には，一致した項目の ***command*** を設定

TS
Std class

したコマンドイベントを発行します。コマンドが ***cmCommandSetChanged*** のブロードキャストイベントがあった場合には，ステータスラインに再表示を行なわせてホットキーが有効にされたり，無効にされたことを反映させます。

関連項目　***TView::handleEvent***，***TStatusLine::draw***

hint　virtual const char *hint (ushort aHelpCtx) ;

hint は，デフォルト処理として 0 ポインタを返します。場面に即したヒント文字列として引数 *aHelpCtx* を使用したい場合には，この関数をオーバーライドする必要があります。0 でない文字列が指定された場合には，その文字列がステータスラインに区切りを示す縦線に続けて表示されます。

関連項目　***TStatusLine::draw***

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream*** クラス

update　void update ( ) ;

ステータスラインを更新します。現在のヘルプコンテキストにしたがって，***defs*** で指定されるリストから適当な項目を選択した後で，項目に変更があった場合には，***drawView*** を呼び出してステータスラインの再表示を行ないます。

関連項目　***TStatusLine::defs***

write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

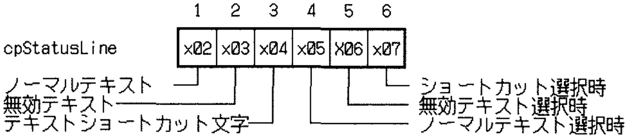
関連項目　***TStreamableClass***，***TStreamable***，***opstream*** クラス

関連する関数

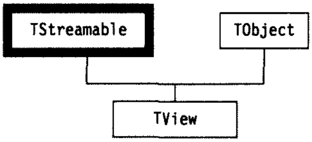
演算子関数で ***TStatusLine*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

ステータスラインはデフォルトパレット ***cpStatusLine*** を使用し，標準アプリケーションパレットのエントリ 2~7 に割り付けます。



TStreamable　　TOBJSTRM.H



TView は，***TObject*** と抽象クラス ***TStreamable*** の 2 つを基本クラスとしています。ビューとなるクラスはすべて，究極的には ***TView*** から導出されているので，***TStreamable*** を継承していることになります。***TCollection***，***TStrListMaker***，***TStringList***，***TxxxInit*** などのビューとならないクラスも，***TStreamable*** を基本クラスとしています。これらのクラスのオブジェクトは，Turbo Vision のストリームマネージャを使用してストリームとの間で読み書きできることから，これらのクラスは「ストリーム入出力可能

である」といいます。

独自にストリーム入出力可能なクラスを開発する場合には，その上位クラスのどれかが必ず ***TStreamable*** となるようにしてください。そのためには，すでにストリーム入出力可能であるクラスを基本クラスにするという方法が簡明です。

TStreamable は抽象クラスなので，インスタンス化を行なうことはできません。ストリーム入出力可能なクラスをストリームとの間で読み書きできるようにするためには，そのクラスにおいて ***streamableName***, ***read***, および ***write*** の 3 個の純粋仮想関数をオーバーライドする必要があります。

メンバ関数

read virtual void *read (ipstream& is) = 0; protected

これは純粋仮想関数であり，すべての導出クラスでオーバーライドする（あるいは純粋仮想として再定義する）必要があります。ストリーム入出力可能な各クラスにおいてオーバーライドする ***read*** 関数では，***ipstream*** オブジェクト ***is*** から必要なデータメンバを読み出してくるようにしなければいけません。一般には，まず基本クラスの ***read*** を呼び出し（***read*** がある場合），それからその導出クラスで追加されたデータメンバがあれば，それを読み出してくるようにします。

関連項目 ***ipstream***, ***TStreamableClass***

streamableName virtual const char *streamableName () const = 0; private

TStreamable にはコンストラクタがありません。この関数は，すべての導出クラスでオーバーライドする（あるいは純粋仮想として再定義する）必要があります。この関数を呼び出してきたオブジェクトのクラス名を返すためのものです。この名前は，ストリームマネージャがストリームを登録する際に使用します。***TView*** では，この関数を次のようにオーバーライドしています。

```
virtual const char *streamableName() const { return TViewName; }
```

TView::TViewName は，"TView" という名前の入っている静的な文字配列です。返される名前は，0 で終わる一義的な文字列にする必要があります。そのためには，ストリーム入出力可能なクラスの名前を使用すれば一番確実です。

関連項目　***TStreamableClass***，***opstream***，***ipstream***

write　virtual void write (opstream& os) = 0；　protected

これは純粋仮想関数であり，すべての導出クラスでオーバーライドする（あるいは純粋仮想として再定義する）必要があります。ストリーム入出力可能な各クラスにおいてオーバーライドする ***write*** 関数では，***opstream*** オブジェクト *os* に必要なデータメンバを書き込むようにしなければいけません。一般には，まず基本クラスの ***write*** を呼び出し（***write*** がある場合），それからその導出クラスで追加されたデータメンバがあれば，それを書き込むようにします。

関連項目　***TStreamableClass***，***opstream***

フレンド

opstream および ***ipstream*** の 2 つのクラスは，***TStreamable*** のフレンドです。したがって，それらのクラスのメンバ関数は，すべて ***TStreamable*** のプライベートメンバにアクセスすることができます。

TS
Std clas

TStreamableClass　　TOBJSTRM.H

TStreamableClass は，***TStreamableTypes*** と ***pstream*** がストリーム入出力可能なクラスの登録を行なう際に使用します。

メンバ関数

コンストラクタ　TStreamableClass (const char *n, BUILDER b, int d);

指定した名前とビルダ関数を持つ ***TStreamable*** オブジェクトを作成した後, ***registerType*** を呼び出します。ストリーム入出力可能なクラス ***TClassname*** には,それぞれ ***build*** メンバ関数が用意されます。またストリーム入出力可能な各クラスとリンクされるストリーム I/O では, メンバではありませんが一般に使用されるオーバーロード演算子 >> および << を使用することができます。型を保障したオブジェクトのストリーム I/O を行なうためには, ストリームマネージャは各クラスの名前と型情報をアクセスする必要があります。どのようなアプリケーションに対しても確実に適切な関数をリンクさせるようにするためには, 以下のような外部参照を行なう必要があります。

```
extern TStreamableClass registerTClassName;
```

ここで, ***TClassName*** はストリームされるべきオブジェクトのクラス名です（***registerTClassName*** は1個の識別子です)。この関数は, ***TClassName*** を登録する(どの ***build*** 関数を使用すべきかをストリームマネージャに知らせる) だけではなく, 依存するクラスもすべて自動的に登録します。1つのクラスを複数回登録しても, 障害やオーバーヘッドは何も生じません。

BUILDER は, 以下のように typedef 宣言されています。

```
typedef TStreamable *(*BUILDER)();
```

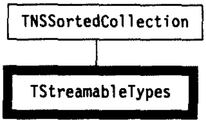
関連項目

TStreamable, ***TStreamableTypes***, ***read***, ***write***, ***build***, ***TStreamableTypes::registerTypes***, ***ipstream***, ***opstream***

フレンド

TStreamableTypes, *opstream* および *ipstream* の各クラスは, *TStreamableClass* のフレンドです。したがって, それらのクラスのメンバ関数は, すべて *TStreamableClass* のプライベートメンバにアクセスすることができます。

TStreamableTypes　　　　TOBJSTRM.H



TStreamableTypes は, *TNSSortedCollection* から private に導出されたクラスです。アプリケーションが使用するストリーム入出力可能な型をすべて登録してあるデータベースを管理します。*opstream*, *ipstream* は, このデータベースを参照して, オブジェクトに合った適当な *read* 関数, *write* 関数を使用します。プライベートに導出されたクラスなので, 継承されたメンバはすべて *TStreamableTypes* 内のプライベートメンバとなります。

メンバ関数

コンストラクタ　TStreamableTypes ( ) ;

基本クラスの *TNSCollection* コンストラクタを呼び出し, *TStreamableTypes* コレクションを作成します。

関連項目　*TNSCollection::TNSCollection*

デストラクタ　~TStreamableTypes ( ) ;

コレクションの *limit* を 0 にします。コレクションの消去は行な

いません（***shouldDelete*** データメンバが *False* にされているため)。

関連項目　***TNSCollection::~TNSCollection***, ***TNSCollection::shouldDelete***

lookup　const TStreamableClass *lookup (const char *name) ;

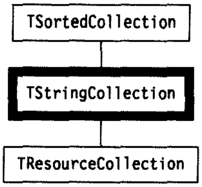
コレクションにおいて，引数 *name* に対応するクラスへのポインタを返します。なければ 0 を返します。

registerTypes　void registerTypes (const TStreamableClass *d) ;

d をコレクションに挿入することによって，引数のクラスを登録します。

関連項目　***TNSCollection::insert***, ***TStreamableClass***

TStringCollection　　RESOURCE.H



TStringCollection は ***TSortedCollection*** の単純な導出クラスであり，ASCII 文字列を要素とするソート済みのリストです。***TStringCollection::compare*** はオーバーライドされ，ASCII 文字列を通常の辞書順に並べ替えるようになっています。***compare*** をオーバーライドして英語以外の文字セットの並べ替えを行なわせるようにすることもできます。

メンバ関数

コンストラクタ　TStringCollection (short aLimit, short aDelta) ;

指定した値を持つ ***TStringCollection*** オブジェクトを作成します。

コンストラクタ TStringCollection (StreamableInit streamableInit) ;

protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目 ***TSortedCollection::TSortedCollection***

build static TStreamable *build ( ) ;

ストリーム読み出しを行なう際，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***, ***ipstream::readData***

compare virtual int compare (void *key1, void *key2) ;

「文字列」key1, key2 を比較します。*key1* < *key2* なら <0 を，*key1*=*key2* なら 0 を，*key1* > *key2* なら >0 を返します。

関連項目 ***TSortedCollection::search***

freeItem virtual void freeItem (void *item) ;

ソート済みコレクションから文字列 *item* を削除し，その文字列を廃棄します。

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***, ***TStreamable***, ***ipstream***

readItem void *TStringCollection::readItem (ipstream& is) ;

コレクション内の要素ごとに呼び出されます。***TCollection*** または

TS
Std class

TSortedCollection から導出したすべてのクラスで，この種の関数をオーバーライドして要素を正しく読み出すようにする必要があります。***TSortedCollection*** では，すでにこの関数をオーバーライドしています。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream***

write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目　***TStreamableClass***，***TStreamable***，***opstream***

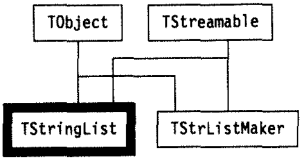
writeItem　void TStringCollection::writeItem (void * obj, opstream& os) ;

コレクション内の要素ごとに呼び出されます。***TCollection*** または ***TSortedCollection*** から導出したすべてのクラスでは，この種の関数をオーバーライドして要素を正しく書き込むようにする必要があります。***TSortedCollection*** では，すでにこの関数をオーバーライドしています。

関連項目　***TStreamableClass***，***TStreamable***，***opstream***

関連する関数

演算子関数で ***TStringCollection*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。



TStringList は，ストリームとの間で文字列のやりとりを行なうためのクラスです。文字列リスト中の各文字列は，0～65,535 までの一義的な番号で識別されます。文字列リストは，文字列をメモリ中にではなくストリームに格納するので，通常の文字列リテラルよりもメモリ消費が少なくて済みます。また文字列リストの場合，文字列をプログラム中に組み込むわけではないので，多国語化を簡単に実現することができます。

TStringList は，文字列にアクセスするためのメンバ関数しか用意していません。文字列リストは，***TStrListMaker*** を使用して作成する必要があります。

TStrIndexRec

文字列リストに関しては，***TStrIndexRec*** という小さなクラスが使用されます。***TStrIndexRec*** は，RESOURCE.H で以下のように定義されています。

```
class TStrIndexRec
{

public:

   TStringIndexRec();        // コンストラクタは count=0 とします。

   ushort key;
   ushort count;
   ushort offset;
}
```

TStringList のデータメンバ ***index*** は，***TStrIndexRec*** オブジェクトへのポインタです。

TS
Std class

メンバ関数

コンストラクタ TStringList (StreamableInit streamableInit) ; protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目 ***TStrListMaker::TStrListMaker***, ***TStringList::get***

デストラクタ ~TStringList ( ) ;

その文字列リストに割り当てられていたメモリを解放します。

関連項目 ***TStrListMaker::TStrListMaker***, ***~TStringList***

build static TStreamable *build ( ) ;

ストリーム読み出しを行なう際，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***, ***ipstream::readData***

get void get (char *dest, ushort key) ;

dest に *key* で指定された文字列を返します。指定された *key* を持つ文字列がない場合には，空の文字列を返します。

関連項目 ***TStrListMaker::put***

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***, ***TStreamable***, ***ipstream*** クラス

write virtual void write (opstream& os) ;

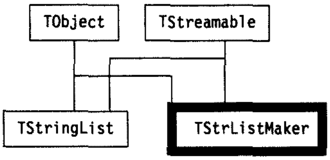
出力ストリーム *os* への書き込みを行ないます。

関連項目　*TStreamableClass*, *TStreamable*, *opstream* クラス

関連する関数

演算子関数で ***TStringList*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TStrListMaker　　RESOURCE.H



TStrListMaker は，***TStringLis***t が使用する文字列リストを作成するための単純なオブジェクト型です。

メンバ関数

コンストラクタ　TStrListMaker (ushort aStrSize, ushort aIndexSize) ;

サイズが *aStrSize* で，*aIndexSize* 個のインデックスを持つ文字列リストをメモリ上に作成します。指定されたサイズの文字列バッファとインデックスバッファはヒープ上に割り当てられます。

aStrSize には，文字列リストに追加されるすべての文字列を格納できるだけの充分な大きさを指定する必要があります（各文字列は，文字列長＋終端の 0 の分のメモリサイズを使用します)。

文字列を文字列リストに追加するとき（***TStrListMaker::put*** を

TS
Std class

使用して)，文字列インデックスが作られます。

キー番号が連続する文字列がある場合は，最高 16 個までは 1 つのインデックスレコードに記録されます。

aIndexSize には，文字列が追加されるときに生成されるインデックスレコードを収容できるだけの充分な大きさを指定する必要があります。各インデックスには，6 バイトずつ使用されます。

コンストラクタ　TStrListMaker (StreamableInit streamableInit)； protected

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TStringList::put***，***~TStrListMaker***

デストラクタ　~TStrListMaker ( )；

文字列リストメーカに割り当てられていたメモリを解放します。

関連項目　***TStrListMaker::TStrListMaker***

build　static TStreamable *build ( )；

ストリーム読み出しを行なう際，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***，***TStreamable***

put　void put (ushort key, const char *s)；

指定された文字列 *s* を，呼び出している文字列リストに指定されたキー番号で追加します。

write　virtual void write (opstream& os)；

出力ストリーム *os* への書き込みを行ないます。

関連項目　*TStreamableClass*, *TStreamable*, *opstream*

関連する関数

演算子関数で ***TStrListMaker*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TSystemError　　　　SYSTEM.H

TSystemError

TSystemError は，システムエラーハンドラとそれに関連するサービスを用意します。ほとんどのメンバがプライベートであり，Turbo Vision の通常のアプリケーションであれば直接関与することはありません。

データメンバ

ctrlBreakHit　static Boolean near ctrlBreakHit ;

プログラムの実行中に ***STOP*** キー入力があれば，***True*** にされます。

メンバ関数

コンストラクタ　TSystemError ( ) ;

TSystemError オブジェクトを作成し，***resume*** を呼び出してシステムエラーハンドラをインストールします。

デストラクタ　~TSystemError ( ) ;

suspend を呼び出して，システムエラーハンドラを除去します。

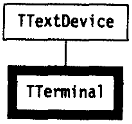
resume　static void resume ( ) ;

システムエラーハンドラをインストールします。

suspend　static void suspend ( ) ;

システムエラーハンドラを除去します。

TTerminal　　TEXTVIEW.H



TTerminal は，文字列をバッファ処理しながら読み書きを行なう「ダム」端末を実現します。デフォルトで，64 K バイトのリングバッファを使用します。

データメンバ

buffer　char *buffer ;

その端末のバッファの第 1 バイト目を指すポインタ。

bufSize　ushort bufSize ;

その端末のバッファのバイト単位でのサイズ。

queBack　ushort queBack ;

その端末のバッファに格納されている最後のバイトを示すオフセット（バイト単位での)。

queFront　ushort queFront ;

その端末のバッファに格納されている 1 番目のバイトを示すオフセ

ット（バイト単位での）。

メンバ関数

コンストラクタ　TTerminal (const TRect& bounds, TScrollBar *aHScrollBar, TScrollBar *aVScrollBar, ushort aBufSize) ;

まず，指定された *bounds* とスクローラを引数に指定して ***TTextDevice::TTextDevice*** を呼び出し，次に ***bufSize*** が *aBufSize* のバッファ（***buffer*** をそのバッファへのポインタとする）を作成することにより，指定された *bounds*，水平，垂直両方のスクロールバー，およびバッファを備える ***TTerminal*** オブジェクトを作成します。***growMode*** には，***gfGrowHiX*** | ***gfGrowHiY*** を設定します。また ***queFront*** と ***queBack*** を，ともに 0 で初期化してバッファを空にします。そしてカーソルをビューの原点（0,0）に表示します。

関連項目　***TScroller::TScroller***，***TTextDevice::TTextDevice***

デストラクタ　~TTerminal ( ) ;

バッファを解放し，***~TTextDevice*** を呼び出します。

関連項目　***~TScroller***，***~TTextDevice***

bufDec　void bufDec(ushort& val);

キューのオフセットの折り返しを行なうために使用されます。*val* が 0 なら，*val* を（***bufSize*** − 1）にします。0 以外のときには，*val* をデクリメントします。

関連項目　***TTerminal::bufInc***

bufInc　void bufInc (ushort& val) ;

キューのオフセットの折り返しを行なうために使用されます。*val* を 1 ずつインクリメントし，その結果 *val* >= ***bufSize*** となったら *val* を 0 にします。

関連項目　***TTerminal::bufDec***

canInsert　Boolean canInsert (ushort amount);

amount で指定された個数のバイトを，先頭の行を捨てることなく端末バッファに挿入できるなら *True* を返します。そうでない場合には *False* を返します。

do_sputn　int do_sputn (const char *s, int);

streambuf クラスにある関数をオーバーライドしています。これは内部的な関数であり，内部バッファに文字列を挿入するときに呼び出されます。

draw　virtual void draw ( );

TTerminal のスクローラを再表示する必要が生じたときに呼び出されます。たとえば，スクロールバーがクリックされたとき，ビューが表に出てきたとき，***delta*** 値が変更されたとき，あるいはテキストの追加によりスクロールが必要になったときなどです。

nextLine　ushort nextLine (ushort pos);

pos で指定された位置以降にある行の始まりを示すバッファオフセットを返します。

関連項目　***TTerminal::prevLines***

prevLines　ushort prevLines (ushort pos, ushort Lines);

pos で指定された位置より *Lines* 行だけ前にある行の始まりを示すバッファオフセットを返します。

関連項目　***TTerminal::nextLine***

queEmpty　Boolean queEmpty ( );

queFront が ***queBack*** と等しい場合に *True* を返します。

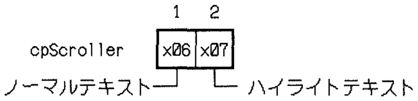
関連項目　***TTerminal::queFront***，***TTerminal::queBack***

フレンド

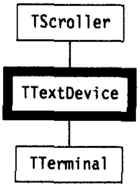
関数 ***genRefs*** は，***TTerminal*** のフレンドです。

パレット

端末オブジェクトは，デフォルトパレット ***cpScroller*** を使用し，標準アプリケーションパレットのエントリ 6~7 に割り付けます。



TTextDevice　　TEXTVIEW.H



TTextDevice は，スクロール可能な TTY 型のテキストビューアー/デバイスドライバです。***TScroller*** から継承したメンバ以外に，***TTextDevice*** は，デバイスとの間で文字列の読み書きを行なうための仮想メンバ関数を定義しています。***TTextDevice*** は，実際の端末ドライバを導出するための基本クラス型としてのみ存在します。***TTextDevice*** は，***TScroller*** のデストラクタを使用します。

メンバ関数

コンストラクタ

TTextDevice (const TRect& bounds,
TScrollBar *aHScrollBar, TScrollBar *aVScrollBar) ;

指定された *bounds* とスクローラを引数に指定して ***TTextScroller::TTextScroller*** を呼び出すことにより，指定された *bounds* と水平，垂直両方のスクロールバーを備える ***TTextDevice*** オブジェクトを作成します。

関連項目 ***TScroller::TScroller***

do_sputn

int do_sputn (const char *s, int) ;

streambuf クラスにある関数をオーバーライドしています。これは内部的な関数であり，内部バッファに文字列を挿入するときに呼び出されます。

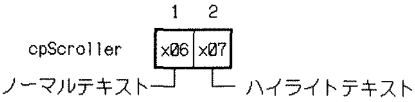
overflow

int overflow(int) ;

streambuf クラスにある関数をオーバーライドしています。***streambuf*** 内にある内部バッファが一杯になっている際，その ***streambuf*** にリンクされている ***iostream*** が，そのバッファにさらに文字を書き込もうとしたときに，***overflow*** が呼び出されます。引数は ***overflow*** を引き起こした文字です。***TTextDevice*** では，その下部構造としてある ***streambuf*** がバッファを持っていないので，文字もすべてオーバーフローを引き起こすことになります。***TTerminal*** などの ***TTextDevice*** の導出クラスでは，このような文字を出力ストリームへの次の文字として扱う必要があります。

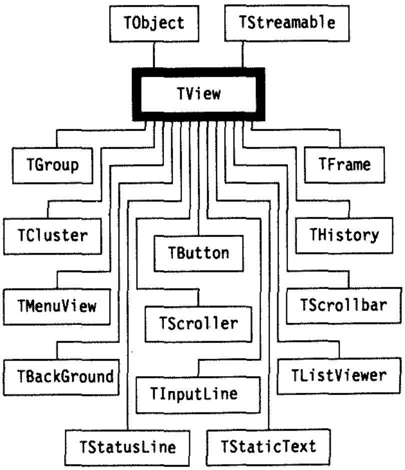
パレット

テキストデバイスオブジェクトはデフォルトパレット ***cpScroller*** を使用し，標準アプリケーションパレットのエントリ 6~7 に割り付けます。



TView　　　　　　　　　　　　　　　　　VIEWS.H

階層図が混雑しないように詳細は省略してあります。完全な階層は100ページに示してあります。



ほとんどのプログラムが，***TView*** の導出オブジェクト（***TFrame***，***TScrollBar***，***TScroller***，***TListViewer***，***TGroup***，***TWindow***）を利用します。

Turbo Vision プログラムで，***TView*** オブジェクトをインスタンス化することはまずありません。***TView*** クラスは，その導出クラスに基本的なデータや機能を用意するためにあります。

データメンバ

commandSetChanged　static Boolean commandSetChanged ;

enableCommand 呼び出し，***disableCommand*** 呼び出し，または

setCommands 呼び出しによって，ビューのコマンドセットが変更されたときに，*True* に設定されます。

関連項目 ***TView::enableCommand(s)***，***TView::disableCommand(s)***，***TView::setCommands***

curCommandSet　static TCommandSet curCommandSet ;

このビューで現在有効なコマンドセットが入ります。最初は，***cmZoom***，***cmClose***，***cmResize***，***cmNext***，***cmPrev*** の各コマンドは無効にされます。***handleEvent*** が受信したコマンドイベントの中で，どれを処理すべきかを決定するためにこのデータメンバを絶えず監視します。***curCommandSet*** 内 は，直接変更しないようにしてください。必要に応じて，***set***，***enable*** または ***disable*** 呼び出しを使用してください。

関連項目　***cmXXXX*** 定数，***TView::setCommands***，***TView::enableCommand(s)***，***TView::disableCommand(s)***

cursor　TPoint cursor ;

ビュー内のハードウェアカーソルの位置。カーソルは，そのビューにフォーカスがあり(***sfFocused***)，かつカーソルがオン(***sfCursorVis***)のときだけ表示されます。カーソルの形は，アンダーラインかブロックのどちらかです（***sfCursorIns*** によって決定される)。

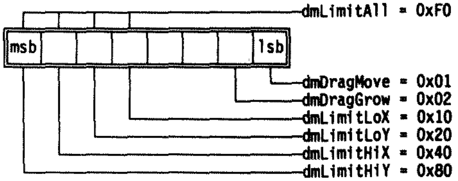
関連項目　***setCursor***，***showCursor***，***hideCursor***，***normalCursor***，***blockCursor***

dragMode　uchar dragMode ;

マウスでドラッグされたときのビューの振舞いを決定します。

dragMode の各ビットの定義は，以下のとおりです。

図 13.1
dragMode の
ビットマップ



dragMode マスクの定義は，第 16 章の「**dmXXXX dragMode 定**

数」に示してあります。

関連項目 ***TView::dragView***

errorAttr static uchar errorAttr ;

無効なパレット選択を意味する属性です。たとえば，***mapColor*** が，引数に無効な色を指定して呼び出されたときに ***errorAttr*** を返します。***errorAttr*** には，デフォルトで，白地に赤のテキストをブリンクさせる 0xCF または 0x47 が設定されます。

関連項目 ***TView::mapColor***

eventMask ushort eventMask ;

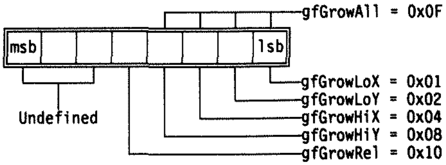
eventMask は，そのビューがどのイベントクラスを受け付けるかを決定するビットマスクです。デフォルトの ***eventMask*** では，***evMouseDown***，***evKeyDown*** および ***evCommand*** が有効にされます。***eventMask*** に 0xFFFF を設定すると，そのビューはすべてのイベントクラスを受け付けるようになります。逆に 0 を設定すると，そのビューは，イベントを何も受け付けなくなります。イベントクラスの詳細については，第 16 章の「**evXXXX イベント定数**」を参照してください。

関連項目 ***TView::handleEvent***

growMode uchar growMode ;

オーナービューにサイズの変更があったときに，そのビューの拡大・縮小をどのようにするかを決定します。***growMode*** には，以下の ***growMode*** マスクのうちの 1 個以上が代入されます。

図 13.2
growMode の
ビットマップ



例 growMode = gfGrowLoX | gfGrowLoY ;

関連項目 ***gfXXXX growMode*** 定数

helpCtx ushort helpCtx ;

ビューのヘルプコンテキスト。そのビューにフォーカスがあるとき，コンテキスト番号が ***hcNoContext*** でないかぎり，このデータメンバがアプリケーションのヘルプコンテキストを表わします。***hcNoContext*** の場合にはヘルプコンテキストはありません。

関連項目 ***TView::getHelpCtx.***

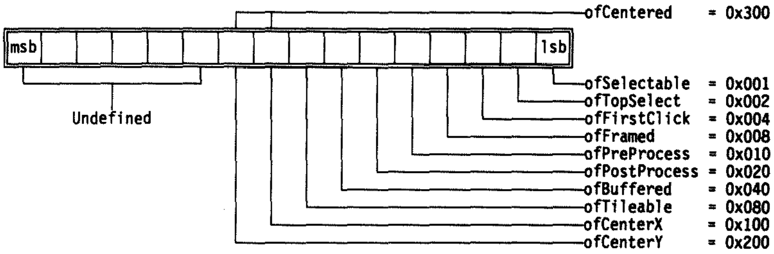
next TView *next ;

Z オーダーで見たときの，次の同列ビューへのポインタ。これが最後のサブビューである場合には，***next*** にはオーナーの 1 番目のサブビューが入ります。

options ushort options ;

ワードフラグ ***options*** は，ビューのいろいろな動作を決定します。***options*** の各ビットの定義は以下のとおりです。

図 13.3
options の
ビットフラグ



options フラグの詳細については，第16章の「**ofXXXX オプションフラグ定数**」を参照してください。また第10章の 278 ページ以降も参照してください。

origin TPoint origin ;

そのビューの左上隅の，オーナーの原点に対する（x, y）相対座標。

関連項目 ***moveTo***, ***locate***

owner TGroup *owner ;

owner は，このビューを所有している ***TGroup*** オブジェクトへのポインタです。0 の場合には，そのビューにオーナーはありません。ビューは，そのオーナービュー内部に表示され，オーナーの境界となる矩形でクリップされます。

showMarkers static Boolean showMarkers ;

フォーカスのあるコントロールの周りにインジケータを配置するかどうかを決めます。***TProgram::initScreen*** が，モノクロのビデオモードのときに ***showMarkers*** を *True* にし，それ以外のときに *False* にします。ただし，必要ならカラーモードおよび白黒モードの場合でもこれをオンにすることができます。

関連項目 ***TProgram::initScreen***，***specialChars***

shutDown virtual void shutDown ( ) ;

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するために内部的に使用します。***shutDown*** は，***destroy*** が呼び出された際，関連するデータメンバの設定を適切に行なえるようにするために多くのクラスでオーバーライドされます。

関連項目 第 6 章『**信頼性の高いプログラムを開発するには**』

size TPoint size ;

ビューのサイズ。

関連項目 ***growTo***，***locate***

state ushort state ;

ビューの状態は，このデータメンバ ***state*** の各ビットのオン/オフによって表わされます。***TView*** のメンバ関数の多くがデータメンバ ***state*** を検査したり，***TView::setState*** を呼び出して ***state*** を変更したりします。

TView::getState (*aState*) は，そのビューの ***state*** が *aState* なら *True* を返します。***state*** の各ビットは，***sfXXXX*** 定数とし

TV
Std clas

てニーモニックで表現されます。*sfXXXX* 定数については，第 16 章の「**sfXXXX 状態フラグ定数**」で説明しています。

メンバ関数

コンストラクタ　TView (const TRect& bounds) ;

指定した *bounds* 矩形領域を備える ***TView*** オブジェクトを作成します。***TView::TView*** は ***TObject*** コンストラクタを呼び出し，新しい ***TView*** オブジェクトのデータメンバに以下の値を設定します。

cursor	(0, 0)
dragMode	***dmLimitLoY***
eventMask	***evMouseDown*** | ***evKeyDown*** | ***evCommand***
growMode	0
helpCtx	***hcNoContext***
next	0
options	0
origin	(*bounds.A.x*, *bounds.A.y*)
owner	0
size	(*bounds.B.x* − *bounds.A.x*, *bounds.B.y* − *bounds.A.y*)
state	***sfVisible***

コンストラクタ　TView (StreamableInit streamableInit) ;　**protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに，vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TObject::TObject***

デストラクタ　~TView ( ) ;

そのビューを画面から消します。さらにそのビューにオーナーがある場合には，グループからそのビューを削除します。

blockCursor void blockCursor ( ) ;

sfCursorIns をセットし，カーソルをブロック形に変更します。カーソルは，同時に ***sfCursorVis*** もセットされているとき(かつそのビューが表示されているとき）のみ表示されます。

関連項目 ***sfCursorIns***，***sfCursorVis***，***TView::normalCursor***，***TView::showCursor***，***TView::hideCursor***

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***，***ipstream::readData***

calcBounds virtual void calcBounds (TRect& bounds, TPoint delta)

オーナーのサイズに変更があったときに，オーナーがすべてのサブビューに対して繰り返し ***calcBounds*** と ***changeBounds*** を呼び出します。***calcBounds*** は，オーナーのサイズが *delta* だけ変更されたという指定に基づいて，そのビューの新しい境界を計算して *bounds* にその新しい境界を返します。

TView::calcBounds は，データメンバ ***TView::growMode*** に設定してあるフラグにしたがって新しい境界を計算します。

関連項目 ***TView::getBounds***，***TView::changeBounds***，***gfXX-XX growMode*** 定数

changeBounds virtual void changeBounds (const TRect& bounds) ;

changeBounds は，そのビューの境界(データメンバ ***origin*** および ***size***)を *bounds* パラメータで指定された矩形に変更します。境界を変更したらビューを再表示します。***changeBounds*** は，***TView*** のいろいろなメンバ関数から呼び出されますが直接には呼び出さないでください。

TView::changeBounds は，まず ***setBounds***(*bounds*) を呼び出し，次に ***drawView*** を呼び出します。

関連項目　***TView::locate***, ***TView::moveTo***, ***TView::growTo***

clearEvent　void clearEvent (TEvent& event) ;

handleEvent が，そのイベントをうまく処理できたことを表わすために使用する標準的なメンバ関数です。*event.what* を ***evNothing*** にし，*event.message.infoPtr* を this にします。

関連項目　***handleEvent*** メンバ 関数

commandEnabled　static Boolean commandEnabled (ushort command) ;

指定された ***command*** が現在有効であれば *True* を返します。そうでなければ *False* を返します。モードつき状態を変更するとき，コマンドを任意に有効にしたり，無効にしたりすることができます。以前のモードつき状態に戻る場合には，元のコマンドセットが復元されます。

関連項目　***TView::disableCommand***, ***TView::enableCommand***, ***TView::setCommands***

containsMouse　Boolean containsMouse (TEvent& event) ;

これを呼び出したビューの内側でマウスイベントが発生したのであれば，*True* を返します。そうでなければ，*False* を返します。

関連項目　***mouseInView***

dataSize　virtual ushort dataSize ( ) ;

dataSize は, ***setData*** および ***getData*** がデータレコードとの間で読み書きしたデータのサイズを返します。データレコードの機能は通常，ダイアログボックスのコントロールとなるビューだけが使用します。

TView::dataSize は, データ転送が何もなかった場合には, 0 を返します。

関連項目　***TView::getData***, ***TView::setData***

disableCommand

static void disableCommand (ushort command);

指定されたコマンドを無効にします。この呼び出しによってコマンドセットが変更される場合には，***commandSetChanged*** を *True* にします。

関連項目　***TView::enableCommand***, ***TView::disableCommands***

disableCommands

static void disableCommands (TCommandSet& commands);

引数 *commands* に指定されたコマンド群を無効にします。

関連項目　***TView::commandEnabled***, ***TView::enableCommands***, ***TView::setCommands***

dragView

```
virtual void dragView ( TEvent& event, uchar mode,
                        TRect& limits, TPoint minSize,
                        TPoint maxSize );
```

指定されたドラッグモードにしたがってビューをドラッグします。ドラッグモードは，引数 *mode* に ***dmXXXX*** フラグを設定することによって指定されます。*limits* は，そのビューが移動できる範囲（オーナーの座標系による矩形領域）を指定します。また *minSize* と *maxSize* は，そのビューが拡大・縮小するときのサイズの最大値，最小値を指定します。ドラッグ動作を引き起こすイベントは，*event* 中で，マウスドラッグとカーソルキーの使用とを区別しておく必要があります。

関連項目　***TView::dragMode***, ***dmXXXX*** ドラッグモード定数

draw

virtual void draw ();

ビューが自分自身を表示する必要がでてきたときに呼び出されます。***draw*** は，そのビューの全領域をカバーしなければなりません。***draw*** は，それぞれの導出クラスに合わせてオーバーライドする必要があります。表に出ているビューのみ，すなわち画面に現われている部分だけの表示を行なう ***drawView*** を使用したほうが効率

的であるため，***draw*** を直接呼び出す必要はほとんどありません。必要なら，再表示の必要な矩形領域を知るために ***getClipRect*** を呼び出し，その領域だけを表示するようにすることもできます。これによって，複雑なビューに対する表示性能をかなり改善することができます。

関連項目　***TView::drawView***

drawCursor　void drawCursor ( ) ;

ビューが ***sfFocused*** なら，***resetCursor*** を呼び出してカーソルをリセットします（初期状態に戻します）。

関連項目　***TView::resetCursor***

drawHide　void drawHide (TView *lastView) ;

drawCursor を呼び出し，つづいて ***drawUnderView*** を呼び出します。***drawUnderView*** は，指定の *lastView* に達するまですべてのサブビューを（必要なら影付きで）再表示します。

drawShow　void drawShow (TView *lastView) ;

drawView を呼び出し，***state*** の ***sfShadow*** ビットが立っている場合には，***drawUnderView*** を呼び出して影を表示します。

drawUnderRect　void drawUnderRect (TRect& r, TView *lastView) ;

owner->clip.intersect (*r*)　を呼び出し，表示すべき領域を設定します。そして次のビューから指定された *lastView* までのすべてのサブビューを ***drawSubViews*** を使用して表示します。最後に，***owner−>clip*** を ***owner−>getExtent*** にリセットします。

関連項目　***TView::drawSubViews***

drawUnderView　void drawUnderView (Boolean doShadow,
TView *lastView) ;

drawUnderRect(*r, lastView*) を呼び出します。*r* は，この関数を呼び出しているビューの現在の境界です。*doShadow* が *True* なら，呼び出す前にそのビューの境界を ***shadowSize*** 分広げます。

関連項目　***TView::drawUnderRect***, ***shadowSize***

drawView　void drawView ( ) ;

exposed が *True* を返してきた場合，すなわちそのビューが表に現われている場合（***sfExposed*** を参照）は ***draw*** を呼び出します。***exposed*** が *False* を返してきた場合には何も行ないません。何かを変更したとき，ビューの形態が変更されるためにビューの再表示を行なう必要がある場合には，（***draw*** ではなく）***drawView*** を呼び出す必要があります。

関連項目　***TView::draw***, ***TGroup::redraw***, ***TView::exposed***

enableCommand　static void enableCommand (ushort command) ;

指定のコマンドを有効にします。この呼び出しによってコマンドセットに変更がある場合には，***commandSetChanged*** を *True* にします。

関連項目　***TView::disableCommand***, ***TView::enableCommands***

enableCommands　static void enableCommands (TCommandSet& commands) ;

引数 *commands* に指定されたコマンドをすべて有効にします。この呼び出しによってコマンドセットに変更がある場合には，***commandSetChanged*** を *True* にします。

関連項目　***TCommandSet***, ***commandSetChanged***, ***TView::disableCommands***, ***TView::getCommands***, ***TView::commandEnabled***, ***TView::setCommands.***

endModal　virtual void endModal (ushort command) ;

topView を呼び出して最前面のモードつきビューを探します。そのようなビューがなかった場合（すなわち，***topView*** が 0 を返した場合）には何も行ないません。モードつきビューがあった場合には，そのビューが ***endModal*** を呼び出していくという具合に ***endModal*** の呼び出しが繰り返されます。

結局，***endModal*** は，現在のモードつき状態を終了させます。引数の *command* は，一番最初にモードつき状態を作り出した ***execView*** に渡されます。

関連項目　***TGroup::execView***，***TGroup::execute***，***TGroup::endModal***

eventAvail　Boolean eventAvail ( ) ;

getEvent を呼び出し，イベントがあれば *True* を返します。イベントがあれば ***putEvent*** を呼び出します。

関連項目　***TView::mouseEvent***，***TView::keyEvent***，***TView::getEvent***，***TView::putEvent***

execute　virtual ushort execute ( ) ;

ビューがモードつきになったときに，***TGroup::execView*** が ***execute*** を呼び出してきます。モードつき実行のできるビューにしたい場合には，***execute*** をオーバーライドしてイベントループを組み込む必要があります。***execute*** の返す値が，***TGroup::execView*** の返す値となります。***TView::execute*** は，デフォルト処理として単に ***cmCancel*** を返します。

関連項目　***sfModal***，***TGroup::execute***，***TGroup::execView***，***cmCancel***

exposed　Boolean exposed ( ) ;

ビューのどこかが画面上に現われている場合に *True* を返します。

関連項目　***sfExposed***，***TView::drawView***

getBounds　TRect getBounds ( ) ;

ビューのサイズの現在値として，そのビューの境界となっている矩形をオーナーの座標系を使用して返します。***TRect::a*** を ***origin*** にし，***TRect::b*** を ***origin*** と ***size*** の合計にします。

関連項目　***TView::origin***，***TView::size***，***TView::calcBounds***，***TView::changeBounds***，***TView::setBounds***，***TView::***

getExtent

getClipRect TRect getClipRect ( ) ;

データメンバ ***clip***, すなわち ***draw*** 呼び出しの中で再表示を必要とする最小の矩形領域を返します。***draw*** は, ***getClipRect*** を使用することによって, 複雑なビューに対する表示性能をかなり改善することができます。

関連項目 ***TView::draw***, ***TView::drawView***

getColor ushort getColor (ushort color) ;

そのビューのパレットと, そのすべてのオーナーのパレットをたどり, *color* の下位バイトおよび上位バイトに指定されたパレットインデックスから文字属性に変換します。

関連項目 ***TView::getPalette***, ***TView::mapColor***

getCommands static void getCommands (TCommandSet& commands) ;

現在のコマンドセットを, 引数 *commands* に返します。

関連項目 ***TView::commandsEnabled***, ***TView::enableCommands***, ***TView::disableCommands***, ***TView.setCommands***

getData virtual void getData (void *rec) ;

getData は, *rec* ポインタで指定されたデータレコードにビューから ***DataSize*** バイト分のコピーを行ないます。データレコードの機能は, 通常ダイアログボックスのコントロールとなるビューだけが使用します。***TView::getData*** は, デフォルトでは何も行ないません。

関連項目 ***TView::dataSize***, ***TView::setData***

getEvent virtual void getEvent (TEvent& event) ;

次に有効なイベントを引数 *event* に返します。イベントが何もなければ ***evNothing*** を返します。デフォルトでは, そのビューのオーナーの ***getEvent*** を呼び出します。

関連項目　***TView::eventAvail***，***TProgram::idle***，***TView::handleEvent***，***TView::putEvent***

getExtent　TRect getExtent ( ) ;

ビューの矩形領域を返します。***TRect::a*** に (0, 0) を，***TRect::b*** に ***size*** を設定します。

関連項目　***TView::origin***，***TView::size***，***TView::calcBounds***，***TView::changeBounds***，***TView::setBounds***，***TView::getBounds***

getHelpCtx　virtual ushort getHelpCtx ( ) ;

そのビューのヘルプコンテキストを返します。デフォルトの ***TView::getHelpCtx*** は，***helpCtx*** データメンバの値を，あるいはそのビューがドラッグされている場合には (***sfDragging*** を参照) ***hcDragging*** を返します。

関連項目　***helpCtx***

getPalette　virtual TPalette& getPalette ( ) const ;

getPalette は，そのビューのパレットを表わす文字列を返します。戻り値が 0 ポインタの場合には，そのビューにパレットがないことを意味します。***writeChar*** および ***writeStr*** が，パレットインデックスを物理文字属性に変換するときに ***getPalette*** を呼び出してきます。デフォルトの戻り値は 0 であり，その場合このビューでは何も色変換を行なわないことになります。***getPalette*** は，ほとんどの導出クラスでオーバーライドされます。

関連項目　***TView::getColor***，***TView::mapColor***，***writeXXX***

getState　Boolean getState (ushort aState) ;

aState で指定された状態が，データメンバ ***state*** において設定されている場合には *True* を返します。

関連項目　***state***，***TView::setState***

growTo　void growTo (short x, short y) ;

TView::locate を呼び出し，ビューを指定のサイズに拡大または縮小します。

関連項目　***TView::origin***，***TView::size***，***TView::locate***，***TView::moveTo***

handleEvent　virtual void handleEvent (TEvent& event) ;

handleEvent は，Turbo Vision のすべてのイベント処理を実現するための中心的メンバ関数です。*event* パラメータのデータメンバの中，*what* にはイベントクラス（***evXXXX***）が入り，残りのデータメンバにはそのイベントを補足説明する情報が入ってきます。***handleEvent*** は，イベントの処理を終了したことを示すために ***clearEvent*** を呼び出す必要があります。***handleEvent*** は，ほとんどの導出クラスでオーバーライドされます。

TView::handleEvent は，***evMouseDown*** イベントを受け取ったら次のような処理を行ないます。そのビューが選択（***sfSelected***）されておらず，かつ無効（***sfDisabled***）でない場合に，選択可能（***ofSelectable***）であるならそのビューは，***Select*** を呼び出して自分自身を選択します。***TView::handleEvent*** は，それ以外のイベントは処理しません。

関連項目　***TView::clearEvent***

hide　void hide ( ) ;

setState を呼び出し，***state*** の ***sfVisible*** フラグをクリアすることによって，そのビューを画面から消します。

関連項目　***sfVisible***，***TView::setState***，***TView::show***

hideCursor　void hideCursor ( ) ;

state の ***sfCursorVis*** ビットをクリアすることによって，カーソルを画面から消します。

関連項目　***sfCursorVis***，***TView::showCursor***

keyEvent　void keyEvent (TEvent& event) ;

変数 *event* に，次の ***evKeyDown*** イベントを返します。他のすべてのイベントは無視し，キーボードイベントが得られるまで待ちます。

関連項目　***TView::getEvent***，***TView::eventAvail***

locate　void locate (TRect& bounds)；

ビューの境界を，引数 *bounds* で指定されたものに変更します。新しい位置でビューの再表示を行ないます。***locate*** は，***sizeLimits*** を呼び出し指定された境界が有効であることを確認した後，***changeBounds*** を呼び出し，境界を変更してビューを再表示します。

関連項目　***TView::growTo***，***TView::moveTo***，***TView::changeBounds***

makeFirst　void makeFirst ( )；

ビューを，そのオーナーのサブビューリストの先頭に移します。***makeFirst*** は，***putInFrontOf(owner－>first())*** と同等です。

関連項目　***TView::putInFrontOf***

makeGlobal　TPoint makeGlobal (TPoint source)；

点座標 *source* をローカル（ビュー）からグローバル（画面）に変換し，変換結果を返します。

関連項目　***TView::makeLocal***

makeLocal　TPoint makeLocal (TPoint source)；

点座標 *source* をグローバル（画面）からローカル（ビュー）に変換し，変換結果を返します。***evMouse*** イベントの ***event.where*** データメンバをグローバル座標からローカル座標にするときに利用できます。たとえば，mouseLoc = makeLocal (eventWhere) とします。

関連項目　***TView::MakeGlobal***，***TView::mouseInView***

mapColor　uchar mapColor (uchar color)；

指定された *color* の現在のパレットにおけるオフセットを求めます。***mapColor*** は，グループオーナーのチェインを遡って ***getPalette*** を呼び出していきます。最終的なオーナーのパレットにたどり着くまで，各パレットのオフセットを次々と対応させていきます。チェイン上のすべてのパレットで *color* が無効だった場合（たとえば範囲外だった場合）には，***mapColor*** は ***errorAttr*** を返します。

関連項目　***TView::getPalette***，***TView::errorAtrr***

mouseEvent　Boolean mouseEvent (TEvent& event, ushort mask)；

次のマウスイベントを引数 *event* に入れます。このイベントが引数 *mask* に入っていた場合には，*True* を返します。***evMouseUp*** イベントが発生した場合にも *False* を返します。マウスボタンが押されている間，たとえばテキストエディタでドラッグによるブロックマークを行なっている場合など，このメンバ関数によってマウスを追跡することができます。以下のコードは ***handleEvent*** ルーチンの中，ビューのカーソルをマウスの動きに追随させている部分です。

```
void TMyView::handleEvent(TEvent& event)
{
   TView::handleEvent(event);
   switch ( event.what )
   {
      case evMouseDown:
         repeat
         {
            makeLocal(event.where, mouse);
            setCursor(mouse.x, mouse.y);
         }
         until !(mouseEvent(event, evmouseMove));
         clearEvent(event);
         break;
   }
    ...
};
```

関連項目　***evXXX*** イベントマスク，***TView::keyEvent***，***TView::getEvent***

mouseInView　Boolean mouseInView (TPoint mouse)；

引数 *mouse*（グローバル座標を使用）が，この関数を呼び出しているビュー内にあれば，*True* を返します。

関連項目　***TView::MakeLocal***，***TView::MakeGlobal***

moveTo　void moveTo（short x, short y）;

origin をオーナービューの相対座標（x, y）に移します。ビューのサイズは変えません。

関連項目　***TView::origin***，***TView::size***，***TView::locate***，***TView::growTo***

nextView　TView *nextView（）;

オーナーのサブビューリスト中の次のサブビューへのポインタを返します。そのビューがオーナーリストの最後のビューだった場合には，0 を返します。

関連項目　***TView::prevView***，***TView::prev***，***TView::next***

normalCursor　void normalCursor（）;

state の ***sfCursorIns*** ビットをクリアし，カーソルをアンダーライン型にします。***sfCursorVis*** が立っている場合には，新しいカーソルを表示します。

関連項目　***sfCursorIns***，***sfCursorVis***，***TView::hideCursor***，***TView::blockCursor***，***TView::hideCursor***

prev　TView *prev（）;

オーナーのサブビューリスト中の１つ手前のサブビューへのポインタを返します。そのビューがオーナーリストの最初のビューだった場合には，リスト中の最後のビューを返します。***TView::prev*** はリストを循環リストとして扱いますが，***TView::prevView*** 線形はリストとして扱います。

関連項目　***TView::nextView***，***TView::prevView***，***TView::next***

prevView　TView *prevView（）;

オーナーのサブビューリスト中の1つ手前のサブビューへのポインタを返します。そのビューが，オーナーリストの最初のビューだった場合には0を返します。***TView::prev*** はリストを循環リストとして扱いますが，***TView::prevView*** は線形リストとして扱います。

関連項目　***TView::nextView***，***TView::prev***

putEvent　virtual void putEvent (TEvent& event)；

event で指定されたイベントをイベントキューに入れます。その結果，***getEvent*** はそのイベントを次に返すことになります。この方法による場合，イベントが1個だけイベントキューにプッシュされます。ビューがコマンドイベントを発行するときに，よくこの関数を使用します。たとえば，以下のようにします。

```
event.what = evCommand;
event.command = cmSaveAll;
event.infoPtr = 0;
putEvent(event);
```

TView::putEvent は，デフォルト処理としてそのビューのオーナーの ***putEvent*** を呼び出します。

関連項目　***TView::eventAvail***，***TView::getEvent***，***TView::handleEvent***

putInFrontOf　void putInFrontOf (TView *target)；

この関数を呼び出しているビューを，オーナーのサブビューリスト中で *target* ビューの1つ手前に移します。

```
MyView.putInFrontOf(owner->first);
```

は，*MyView::makeFirst* と等価です。このメンバ関数は，サブビューリスト内のポインタを書き換えていきます。***putInFrontOf*** を実行すると，他のビューの位置および表示されるかどうかの状態に応じて下になるビューは（部分的に）隠されたり（クリップされたり）します。選択可能である（***ofSelectable*** を参照）ビューが，他のすべてのサブビューより前にもってこられた場合には，そのビューが

選択されます。

関連項目　***TView::makeFirst***

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***, ***TStreamable***, ***ipstream*** クラス

resetCursor　virtual void resetCursor ( ) ;

ハードウェアカーソルをリセットします。

select　void select ( ) ;

ビューを選択します (***sfSelected*** を参照)。そのビューのオーナーにフォーカスがある場合には，(***TView::makeFirst*** を呼び出して)そのビューもフォーカスを得ます(***sfFocused*** を参照)。ビューの ***options*** データメンバの ***ofTopSelect*** フラグが立っている場合には，(***TView::makeFirst*** を呼び出して) そのビューをオーナーのサブビューリストの先頭に移します。

関連項目　***sfSelected***, ***sfFocused***, ***ofTopSelect***, ***TView::makeFirst***

setBounds　void setBounds (const TRect& bounds) ;

ビューの境界を決めている矩形を，パラメータ *bounds* に指定されている値にします。データメンバ ***origin*** に *bounds.a* を設定し，データメンバ ***size*** に *bounds.b* と *bounds.a* の差を設定します。***setBounds*** は，オーバーライドした ***changeBounds*** メンバ関数が使用するだけの目的のメンバ関数です。決して直接 ***setBounds*** を呼び出さないようにしてください。

関連項目　***TView::origin***, ***TView::size***, ***TView::calcBounds***, ***TView::changeBounds***, ***TView::getBounds***, ***TView::getExtent***

setCommands　static void setCommands (TCommandSet& commands) ;

現在のコマンドセットを，引数に指定された *commands* に変更し

ます。

関連項目　***TView::enableCommands***，***TView::disableCommands***

setCursor　void setCursor (short x, short y)；

ハードウェアカーソルを，ビュー相対（ローカル）座標での点（ x, y ）にもっていきます。(0,0) が左上隅を意味します。

関連項目　***TView::makeLocal***，***TView::hideCursor***，***TView::showCursor***

setData　virtual void setData (void *rec)；

setData は *rec* で指定されたデータレコードを，***DataSize*** バイト分ビューにコピーします。データレコードの機能は，通常ダイアログボックスのコントロールとなるビューだけが使用します。

TView::setData は，デフォルトでは何も行ないません。

関連項目　***TView::dataSize***，***TView::getData***

setState　virtual void setState (ushort aState, Boolean enable)；

データメンバ ***state*** 内の状態フラグをセットまたはクリアします。*aState* パラメータで変更したい状態フラグを指定し（***sfXXXX*** を参照），*enable* パラメータでそのフラグをオフ（*False*）にするかオン（*True*）にするかを指定します。***setState*** は，次に状態の更新に伴う処理を行ないます。たとえば，そのビューを画面から消した場合に，表に現われてきたビューを再表示したり，カーソルの形が変更された場合（***sfCursorVis*** および ***sfCursorIns***）にハードウェアカーソルを再設定したりします。

setState は，状態フラグに基づいた処理を別途行なわせたい場合にオーバーライドされることがあります。たとえば ***TFrame*** は，以下のリストにあるようにウィンドウが選択されたりドラッグされた場合に，自分自身を再表示できるよう ***setState*** をオーバーライドしています。

```
void TFrame::setState( ushort aState, Boolean enable )
{
```

```
    TView::setState(aState, enable);
    if (aState & (sfActive | sfDragging) != 0)
       drawView();
}
```

また, ***setState*** をオーバーライドする主な理由としては, 以下のように特定のビューが処理するコマンド群を有効にしたり無効にしたりしたいということがあります。

```
void TMyView::setState( ushort aState, Boolean enable )
{ TCommandSet myCommands;
  myCommands.enableCmd(cmCut);
  myCommands.enableCmd(cmCopy);
  myCommands.enableCmd(cmPaste);
  myCommands.enableCmd(cmClear);
  TView::setState( aState, enable );
  if ( aState = sfSelected )
  {
     if enable
        enableCommands(myCommands)
     else
        disableCommands(myCommands)
  }
}
```

関連項目 ***TView::getState***, ***TView::state***, ***sfXXXX*** 状態フラグ

show void show ( ) ;

ビューが ***sfVisible*** なら何も行ないません。***sfVisible*** でない場合に, ***setState*** を呼び出して ***state*** の ***sfVisible*** フラグを立てることによりそのビューを表示します。

関連項目 ***TView::setState***

showCursor void showCursor ( ) ;

sfCursorVis を立てて, ハードウェアカーソルが表示されるようにします。カーソルは, デフォルトでは表示されません。

関連項目 ***sfCursorVis***, ***TView::hideCursor***

sizeLimits virtual void sizeLimits (TPoint& min, TPoint& max);

データメンバ ***size*** がとり得る最大値，最小値を引数 *min*, *max* に入れます。

TView::sizeLimits は，デフォルト処理として *min* には (0, 0) を，*max* には ***owner－＞size*** を返します。***owner*** が 0 の場合には，*max.x* , *max.y* ともに ***MAXSHORT*** にします。

関連項目 ***TView::size***

topView TView *topView ( );

現在のモードつきビューへのポインタを返します。そのようなビューがない場合には 0 を返します。

valid virtual Boolean valid (ushort command);

このメンバ関数は，ビューが構築された後，またはモードつき状態が終了した (***endModal*** 呼び出しによって) ときに，そのビューの有効性をチェックするために使用します。

引数 *command* の値を ***cmValid*** (＝0) とすると，ビューにそのコンストラクタの結果をチェックさせます。つまり，***valid(cmValid)*** は，そのビューが正常に構築され，その時点で使用することができる場合に *True* を返します。そうでない場合には *False* を返します。

引数 *command* にそれ以外 (0 以外) の値を指定した場合，現在のモードつき状態(たとえばモードつきダイアログボックス)が，*command* の値を結果として終了しようとしていることを意味します。この場合には，***valid*** はそのビューの有効性をチェックする必要があります。

ビューが無効だった場合にユーザに警告を発する処理は，***valid*** で行なわなければなりません。

TView::valid は，デフォルトでは単に *True* を返します。

関連項目 ***TGroup::valid***, ***TDialog::valid***, ***TProgram::***

validView

write virtual void write (opstream& os)；

出力ストリーム *os* への書き込みを行ないます。

関連項目 ***TStreamableClass***, ***TStreamable***, ***opstream*** クラス

writeBuf

```
void writeBuf (  short x, short y, short w, short h,
                 const void far *buf )；
void writeBuf (  short x, short y, short w, short h,
                 const TDrawBuffer& buf )；
```

指定されたバッファの内容を，画面の座標 （ *x*, *y* ） から始まる，幅 *w*, 高さ *h* の領域に書き込みます。メンバ関数 ***draw*** からしか使用できません。

buf には，通常 ***TDrawBuffer&*** 型を指定しますが，下位ワードに文字を入れ，上位ワードに属性を入れたロングワード配列を指定することもできます。

関連項目 ***TView::draw***

writeChar

```
void writeChar (  short x, short y, ushort c, uchar color,
                  short count )；
```

文字 *c* を *count* 回コピーしたものを，現在のビューパレットの *color* 番目のエントリで決まる色を使って（ *x*, *y* ） 以降に書き込みます。メンバ関数 ***draw*** しか使用できません。

関連項目 ***TView::draw***

writeLine

```
void writeLine (  short x, short y, short w, short h,
                  const void far *buf )；
void writeLine (  short x, short y, short w, short h,
                  const TDrawBuffer& buf )；
```

バッファ *buf* に格納されている行を，画面の（ *x*, *y* ）から始まる幅 *w*, 高さ *h* の矩形領域に書き込みます。*h* が 1 より大きい場合には，その行を *h* 回繰り返して書き込みます。メンバ関数 ***draw*** からしか使用できません。

関連項目　***TView::draw***

writeStr　void writeStr (short x, short y, const char *str, uchar color) ;

文字列 *str* を，そのビューのパレットの *color* 番目のエントリの色属性を使って (x, y) 以降に書き込みます。メンバ関数 ***draw*** からしか使用できません。

関連項目　***TView::draw***

フレンド

関数 ***genRefs*** は，***TView*** のフレンドです。

関連する関数

演算子関数で ***TView*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

TVMemMgr　　BUFFERS.H

TVMemMgr

セーフティプールの詳細については，第6章，『信頼性の高いプログラムを開発するには』を参照してください。

TVMemMgr は，***TBufEntryList*** およびグローバル演算子 ***new*** とともに，Turbo Vision アプリケーションの低位のメモリ管理を行ないます。***TVMemMgr*** は，特にセーフティプールの管理を行ないます。***TVMemMgr*** および ***TVBufListEntry*** は，たいていのアプリケーションでは舞台裏で機能するオブジェクトなので，アプリケーション側で特別に何か指定したり介在する必要はありません。

データメンバ

safetyPool static void *safetyPool ; private

safetyPool は，セーフティプールメモリ割り当て領域へのポインタです。この値が 0 なら，セーフティプールが割り当てられていないか，セーフティプールを使いきっていることを意味します。

safetyPoolSize static size_t safetyPoolSize ; private

現在のセーフティプールのバイト単位でのサイズです。これはプライベートメンバであり，***TVMemMgr::resizeSafetyPool*** によって更新されます。セーフティプールのデフォルトサイズは，BUFFERS.H で宣言される ***DEFAULT_SAFETY_POOL_SIZE*** 定数で決まります。現在は 8,192 バイトにしてあります。

inited static int inited ; private

セーフティプールの初期化が行なわれたかどうか(成否は別として)を示します。内部的にしか使用されません。

メンバ関数

コンストラクタ TVMemMgr::TVMemMgr ( ) ;

セーフティプールがすでに初期化されていれば（すなわち ***inited*** = 1 なら)何も行ないません。そうでなければ，***resizeSafetyPool*** を呼び出してサイズが定数 ***DEFAULT_SAFETY_POOL_SIZE*** のデフォルトのセーフティプールを確保します。***DEFAULT_SAFETY_POOL_SIZE*** は，現在 BUFFERS.H で 4,096 (バイト) にしてあります。なおコンストラクタは ***inited*** を 1 にします。

allocateDiscardable static void allocateDiscardable (void *&adr, size ↑ t sz) ;

内部的にのみ使用されます。サイズ *sz* のキャッシュバッファ（***TBufListEntry*** オブジェクト）を割り当てようとします。成功

したら，*adr* に割り当てられたバッファへのポインタを返します。失敗したら *adr* を 0 にします。***TGroup::getBuffer*** が，*adr* にデータメンバ ***TGroup::buffer*** を指定して ***allocateDiscardable*** を呼び出してきます。

関連項目　***TVMemMgr::freeDiscardable***，***new*** 演算子，***TGroup::getBuffer***

freeDiscardable　static void freeDiscardable (void *block)；

内部的にのみ使用されます。以前 ***alllocateDiscardable*** 呼び出しで *block* に割り当てられたバッファを解放します。***TGroup::freeBuffer*** が，*block* にデータメンバ ***TGroup::buffer*** を設定して ***freeDiscardable*** を呼び出してきます。

関連項目　***TVMemMgr::allocateDiscardable***，***TGroup::freeBuffer***

resizeSafetyPool　static void resizeSafetyPool (size_t sz = DEFAULT_SAFETY_POOL_SIZE)；

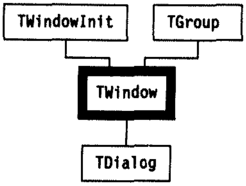
セーフティプールのサイズを *sz* バイトに変更します。それ以前にセーフティプールが使用されていた場合には，まずそれを解放します。***inited*** を 1 にし，***safetyPool*** に新しいセーフティプールへのポインタを設定して，***safetyPoolSize*** を *sz* にします。*sz* 引数を省略した場合には，デフォルト値が使用されます。*sz* に 0 を指定した場合には，***safetyPool*** および ***safetyPoolSize*** を 0 にします。

関連項目　***TVMemMgr*** コンストラクタ，***DEFAULT_SAFETY_POOL_SIZE***

safetyPoolExhausted　static int safetyPoolExhausted ( )；

セーフティプールの初期化が行なわれていて，その割り当てメモリが使い尽くされている場合に 1 (*True*) を返します。それ以外の場合には 0 (*False*) を返します。グローバル関数 ***lowMemory*** が，***safetyPoolExhausted*** を呼び出します。

関連項目　***lowMemory***，***new*** 演算子



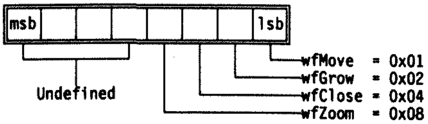
TWindow オブジェクトは通常，***TFrame*** オブジェクトを 1 個，インテリアの ***TScroller*** オブジェクトを 1 個，それに ***TScrollBar*** を 1 個または 2 個備える特殊化されたグループです。これらのリンクされたサブビューによって，***TWindow*** オブジェクトは「目に見えるもの」となります。***TFrame*** オブジェクトは，おなじみの境界であり，タイトルや番号，機能を割り当ててあるアイコン（クローズ，ズーム，ドラッグ）をオプションで持たせることができます。***TWindow*** オブジェクトは，「組み込みの」機能として，マウスドラッグやカーソルキー入力によって移動したり，サイズを変更したりすることができます。しかるべきアイコン領域をマウスクリックすることによって，ズームしたりクローズしたりすることもできます。スクロールバーおよびスクローラの処理方法も「組み込み」です。1 から 9 までの番号の付いたウィンドウは，**GRPH**-*n* キー（ *n* = 1～9 ）で選択することができます。

TWindow は，***TGroup*** とその仮想基本クラスである ***TWindowInit*** を多重継承しています。***TWindowInit*** は，コンストラクタと，フレームオブジェクトを作成し挿入するためのメンバ関数 ***createFrame*** とを用意します。同様の手法が，***THistoryWindow*** と ***THistInit*** の場合にも使用されています。

データメンバ

flags　uchar flags ;

データメンバ ***flags*** は，以下の各ビットの組み合わせが設定されます。



ウィンドウフラグの定義については，第 16 章，「**wfXXXX ウィンドウフラグ定数**」を参照してください。

frame TFrame *frame ;

このウィンドウにリンクされている ***TFrame*** オブジェクトへのポインタ。

関連項目 ***TWindow::initFrame***

number short number ;

このウィンドウに付けられた番号。***TWindow::number*** が 1〜9 のいずれかであれば，その番号がフレームタイトルに表示され，そのウィンドウを **GRPH**-*n* キー（*n* = 1〜9）で選択することができます。

palette short palette ;

ウィンドウが，***wpBlueWindow***，***wpCyanWindow***，***wpGrayWindow*** のうちのどのパレットを使用するかを指定します。デフォルトパレットは ***wpBlueWindow*** です。

関連項目 ***TWindow::getPalette***，***wpXXXX*** 定数

title const char *title ;

フレームに表示されるタイトルを表わす文字列（オプション）。

zoomRect TRect zoomRect ;

ウィンドウの，ズームしていないときの通常の境界。

メンバ関数

コンストラクタ　TWindow (const TRect& bounds, const char *aTitle, short aNumber);

TGroup コンストラクタを呼び出して *bounds* を設定します。デフォルトの ***state*** を ***sfShadow*** に，デフォルトの ***options*** を (***ofSelectable*** | ***ofTopSelect***) に，デフォルトの ***growMode*** を ***gfGrowAll*** | ***gfGrowRel*** に，デフォルトの ***flags*** を (***wfMove*** | ***wfGrow*** | ***wfClose*** | ***wfZoom***) にします。データメンバ ***title*** には *aTitle* を，データメンバ ***number*** には *aNumber* を設定します。デフォルト処理として ***initFrame*** を呼び出し，戻り値の ***frame*** ポインタがゼロ以外であれば，そのフレームをこのウィンドウグループに挿入します。最後に，指定された *bounds* をデフォルトの ***zoomRect*** とします。

コンストラクタ　TWindow (StreamableInit streamableInit);　　**protected**

ストリーム入出力可能なクラスとするためには，そのクラスのオブジェクトに適当なメモリを割り当てるとともに, vtable ポインタを割り付けて初期化するビルダ（組み立て関数）を用意する必要があります。この作業は，***StreamableInit*** 型を引数にしてこのコンストラクタを呼び出すことで行なうことができます。これについては第 8 章も参照してください。

関連項目　***TFrame::initFrame***

デストラクタ　~TWindow ( );

タイトルを消去した後，親のデストラクタを呼び出して，そのウィンドウとすべてのサブビューを廃棄します。

build　static TStreamable *build ( );

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　*TStreamableClass*, *ipstream::readData*

close　virtual void close ( ) ;

valid (*cmClose*) を呼び出します。*True* が返ってきたら，呼び出し元のウィンドウを消去します。

関連項目　***TView::valid***

getPalette　virtual TPalette& getPalette ( ) const ;

palette データメンバのパレットインデックスで指定されているパレット文字列を返します。

関連項目　***TWindow::palette***

getTitle　virtual const char *getTitle (short maxSize) ;

ウィンドウのタイトル文字列である ***title*** を返します。

関連項目　***TWindow::title***

handleEvent　virtual void handleEvent (TEvent& event) ;

まず ***TGroup::handleEvent*** を呼び出してから，***TWindow*** に固有なイベントを以下のようにして処理します。

以下の ***evCommand*** イベントについては，***TWindow::flags*** データメンバでその操作が許可されている場合に処理を行ないます。

cmResize　メンバ関数 ***TView::dragView*** を使用し，ウィンドウの移動またはサイズの変更を行ないます。

cmClose　***cmCancel*** イベントを作成してウィンドウをクローズします。

cmZoom　メンバ関数 ***TWindow::zoom*** を使用してウィンドウをズームします。

keyCode の値が ***kbTab*** または ***kbShiftTab*** の ***evKeyDown*** イベントに対しては，次のサブビューまたは1つ手前のサブビューの選択を行ないます（そのようなサブビューがあった場合）。

command の値が ***cmSelectWindowNum*** の ***evBroadcast*** イ

ベントについては，データメンバ ***event.infoInt*** が ***number*** に等しい場合に，そのウィンドウを選択します。

関連項目　***TGroup::handleEvent***, ***wfXXXX*** 定数, ***TWindow::dragView***, ***TWindow::zoom***

initFrame　virtual void initFrame (TRect) ;

そのウィンドウで使用する ***TFrame*** オブジェクトを作成し，そのフレームへのポインタをデータメンバ ***TWindow::frame*** に格 t 納します。***TWindow*** のコンストラクタが ***initFrame*** を呼び出してきます。***initFrame*** を直接呼び出さないでください。***initFrame*** をオーバーライドし，標準の ***TFrame*** のかわりにユーザ定義のクラスを ***TFrame*** から導出し，インスタンス化することもできます。

関連項目　***TWindow::TWindow***

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamable***, ***ipstream*** クラス

setState　virtual void setState (ushort aState, Boolean enable) ;

まず ***TGroup::setState*** (*aState*, *enable*) を呼び出します。次に *aState* が ***sfSelected*** なら，***setState*** (*sfActive*, *enable*) を呼び出して，ウィンドウとそのすべてのサブビューを有効または無効にします。そして ***cmNext***, ***cmPrev***, ***cmResize***, ***cmClose***, および ***cmZoom*** について ***TView::enableCommands*** または ***TView::disableCommands*** を呼び出します。

関連項目　***TGroup::setState***, ***TView::enableCommands***, ***TView::disableCommands***

shutDown　virtual void shutDown ( ) ;

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するために内部的に使用します。***shutDown*** は，***destroy*** が呼び出されたときに，関連するデータメンバの設定

を適切に行なえるようにするために，多くのクラスでオーバーライドされます。

関連項目　第6章『**信頼性の高いプログラムを開発するには**』

sizeLimits　virtual void sizeLimits (TPoint& min, TPoint& max)；

TView::sizeLimits をオーバーライドします。***TView::sizeLimits*** (*min, max*) を呼び出した後，*min* を最小ウィンドウサイズ（現在（16，6）に初期設定されている）に変更します。

関連項目　***TView::sizeLimits***

standardScrollBar　TScrollBar *standardScrollBar (ushort aOptions)；

そのウィンドウで使用する「標準の」スクロールバーを作成，挿入し，そのスクロールバーへのポインタを返します。「標準」とは，そのスクロールバーが，ウィンドウの角の部分あるいはリサイズアイコンにかぶさることなく，ウィンドウのフレームにぴったり当てはまるという意味です。

aOptions パラメータには，ウィンドウの下枠に沿って配置される水平スクロールバーを設けたい場合には ***sbHorizontal*** を指定し，ウィンドウの右枠に沿って配置される垂直スクロールバーを設けたい場合には，***sbVertical*** を指定します。いずれの場合も，***sbHandleKeyboard*** を併せて指定すれば，マウスクリックだけでなく，キーボードからのカーソル移動キー入力，ページ移動キー入力に対しても，スクロールバーに処理させることができます。

関連項目　***sbXXXX*** スクロールバー定数

write　virtual void write (opstream& os)；

出力ストリーム *os* への書き込みを行ないます。

関連項目　***TStreamableClass***, ***TStreamable***, ***opstream*** クラス

zoom　virtual void zoom ( )；

呼び出し元のウィンドウをズームします。このメンバ関数は通常，***cmZoom*** コマンド（ズームアイコンをクリックしたときに発行さ

れる）を受け付けたときに呼び出されます。

zoom は，呼び出し元のウィンドウとそのオーナーの相対的なサイズ，および ***zoomRect*** の値にしたがってズームを行ないます。

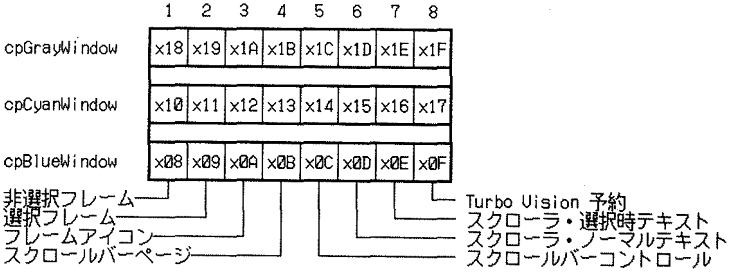
関連項目　***cmZoom***, ***zoomRect***, ***TView::locate***

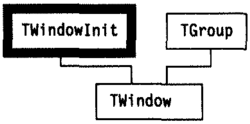
関連する関数

演算子関数で ***TWindow*** に関係するものもありますが，メンバ関数とはなっていません。詳細については 322 ページを参照してください。

パレット

ウィンドウオブジェクトは，デフォルトパレットとして，***cpBlueWindow***（テキストウィンドウ用），***cpCyanWindow***（メッセージ用）および ***cpGrayWindow***（ダイアログボックス用）を使用します。





TWindow は，***TGroup*** とその仮想基本クラスである ***TWindowInit*** を多重継承します。***TWindowInit*** は，コンストラクタとフレームオブジェクトを作成し，挿入するためのメンバ関数 ***createFrame*** とを用意します。同様の手法が，***TProgram***, ***THistoryWindow***，および ***TDeskTop*** の場合にも使用されています。

メンバ関数

コンストラクタ

TWindowInit (TFrame *(* cFrame) (TRect bounds));

引数に関数アドレスをとります。通常 &TWindow::initFrame です。***TWindow*** コンストラクタは，***TGroup***(*bounds*) と ***TWindowInit*** (&TWindow::initFrame) を呼び出し，ウィンドウオブジェクトとそれにリンクされるフレームを作成します。そしてフレームを，ウィンドウグループオブジェクトに挿入します。

createFrame

TFrame *(* createFrame) (TRect bounds);

TWindowInit コンストラクタが，指定の境界を備える ***TFrame*** オブジェクトを作成するために呼び出してきます。***createFrame*** は，作成した ***TFrame*** オブジェクトへのポインタを返します。ポインタが 0 の場合は，この作成作業が失敗したことを示します。

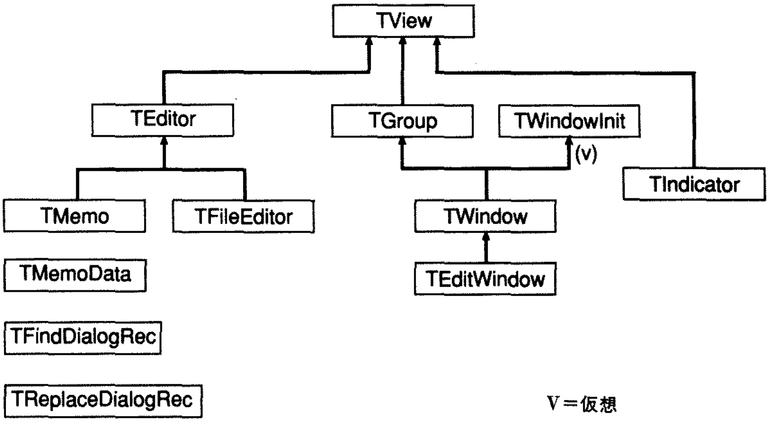
TW
Std class

第 14 章 エディタクラス

TEditor は，***TView*** からの導出クラスであり，Turbo Vision のアプリケーションで使用できる軽くて高速な 64 K バイトのエディタです。***TEditor*** は，マウスのサポート，アンドゥ，クリップボードのカット・コピー・ペースト，自動インデントモード・上書きモード，キー割り当て，検索・置換といった機能を備えています。このエディタは，ファイルを編集したり，ダイアログや入力フォームなどの複数行のメモフィールドとして利用することができます。下の図は，全体の階層を示したものです。

図 14.1
エディタクラス

TEditor は，TView 経由で TStreamable を継承しており，ストリーム入出力可能なクラスとなっています。



TEditor はすべてのエディタの基本クラスで，エディタの機能をほとんど実現します。***TEditor*** のオブジェクトを作成した場合，***TEditor*** はヒープに指定サイズのバッファを割り当てます。バッファは，最初は空です。

TEditor の使用法は，ファイルの編集については TVEDIT.CPP に，メモデータメンバとしては TVFORMS.CPP で説明してあります。これらのファイルは，マスターディスクの DEMOS ディレクトリに入っています。

TEditor のバッファ

TEditor は，**バッファギャップエディタ**です。つまりテキストを2つに分けてセーブします。カーソルよりも前のテキストはバッファの最初に保存され，カーソルよりも後ろのテキストはバッファの最後に保存されます。テキスト間の空白を**ギャップ**といいます。文字を挿入すると，その文字はギャップに挿入されます。エディタは削除されたテキストをギャップに記録し，挿入されたバイト数と削除されたバイト数を管理することでアンドゥをサポートします。アンドゥの実行を求められたら，挿入された文字を削除し，先ほど削除された文字をギャップの最初にコピーして，カーソルをその先ほど削除されたテキストの後ろにもっていきます。

バッファの動作を説明するために，abcdefghijkxxxopqrstuvwxyz という文字を挿入したときの，エディタバッファの図を考えてみましょう。

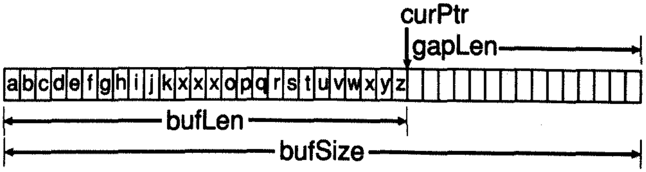
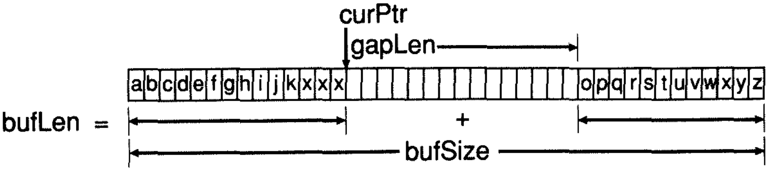


図 14.2
テキスト挿入後のバッファ

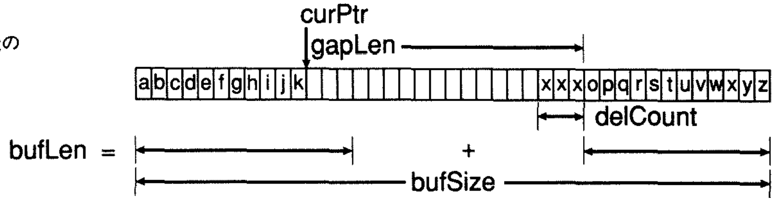
curPtr はカーソルの位置，***gapLen*** はギャップの長さ，***bufLen*** はバッファ内の総文字数です。***bufSize*** はバッファのサイズを示し，常に ***gapLen*** と ***bufLen*** の合計になります。ここでカーソルを xxx のすぐ後ろに移動した場合，バッファは下図のようになります。

図 14.3
カーソル移動後のバッファ



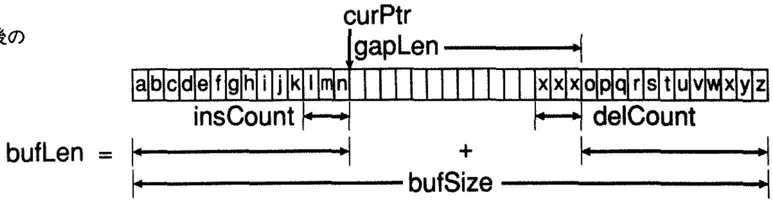
ギャップはカーソルの前にとられていることに注意してください。これにより，テキストを移動しなくても文字をすぐに挿入することができます。ここでバックスペースキーを使って xxx を削除すると，xxx はギャップの最後のほうにコピーされ，カーソルは逆向きに（先頭方向に）移動します。データメンバ ***delCount*** には，削除されたバイト数が記録されます。

図 14.4
xxx を削除した後のバッファ



文字の挿入があると，挿入カウント ***insCount*** がインクリメントされアンドゥで何バイト削除するかを示します。ここで lmn とタイプすると，バッファは以下のようになります。

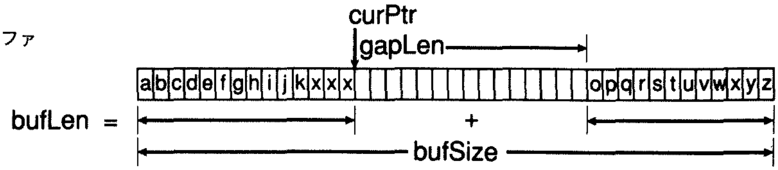
図 14.5
lmn を挿入した後のバッファ



insCount が，挿入された文字のバイト数を表わします。ここで

アンドゥを実行すると lmn は削除され，その先頭から xxx がコピーされ，バッファは編集前の状態に復旧します。

図 14.6
アンドゥ後のバッファ



アンドゥを実行する前にカーソルを移動した場合，ギャップが移動するので，アンドゥ用の情報はすべて消失します。アンドゥは，カーソルを移動してから次にカーソルを移動するまでの間に行なわれた操作だけをやり直します。カーソルの移動があると，それまでに実行した編集作業は「受理された」ものとみなされます。また，アンドゥはバッファ内の領域を使用するので，ユーザがテキストを挿入できなくなることがあります。が，カーソルを移動すれば，その領域を回復することができます。

ブロックの選択

カーソルの前または後ろには，必ず選択部分，すなわちブロックマークがあります。エディタにテキストを挿入した場合，キーを押すか ***insertText*** を使うことで，選択部分の内容は挿入されたテキストで置き換えられます。何も選択していない場合，テキストはそのまま挿入されます。選択部分は，データメンバ ***selStart*** および ***selEnd*** によってマークされます。選択部分は ***setSelect*** 呼び出しで指定することができ，その場合，カーソルも移動します。

オプション

TEditor ではオプションがいくつか用意されており，オプションの状態は ***Boolean*** 型のデータメンバに格納されます。***canUndo*** は，エディタがアンドゥの情報を記録するかどうかを示します。***undo*** は挿入用の領域を一時的に使用するので，アンドゥ機能を無

効にしたほうがよい場合もあるでしょう。アンドゥは，クリップボードに対しては自動的に無効にされます。***overwrite*** は，エディタを上書きモードとするか，挿入モードとするかを示します。***autoIndent*** は**リターン**キーを押したときに，インデントを行ない，前の行のスペース以外の最初の文字の桁にカーソルをもっていくかどうかを表わします。これは，ソースコードの編集を行なうときに便利です。

キー割り当て

ボーランド社の統合開発環境(IDE)で使われているキーと機能の対応付けの多くが，このエディタでもそのままキーに割り当てられています。ただし，ブロックコマンドだけは例外です。***TEditor*** では，持続的なブロックは使わないので，ブロック関係のコマンドは，クリップボードとの間で情報をコピーすることでシミュレートされます。たとえば，***CTRL-K CTRL-B*** はテキストの選択を始めます。***CTRL-K CTRL-K*** はそのテキストをクリップボードにコピーし，そして ***CTRL-K CTRL-C*** はその内容をクリップボードからエディタにペーストします。なおブロックの選択は，***CTRL-K*** の機能割り当てを使うかわりに，シフトキーを押しながらカーソル移動コマンドを実行することでも開始することができます。

これらのキーの機能割り当ては，指定されたキーイベントをコマンドイベントに変換する ***convertEvent*** メンバ関数をオーバーライドすることで変更することもできます。

定数

この節に出てくる定数は EDITORS.H で定義されています。

maxLineLength は 256 に設定されています。

表 14.1
TEditor の定数

項目	値	項目	値
コマンド定数			
cmSave	=80,	*cmPageDown*	=509,
cmSaveAs	=81,	*cmTextStart*	=510,
cmFind	=82,	*cmTextEnd*	=511,
cmReplace	=83,	*cmNewLine*	=512,
cmSearchAgain	=84;	*cmBackSpace*	=513,
cmCharLeft	=500,	*cmDelChar*	=514,
cmCharRight	=501,	*cmDelWord*	=515,
cmWordLeft	=502,	*cmDelStart*	=516,
cmWordRight	=503,	*cmDelEnd*	=517,
cmLineStart	=504,	*cmDelLine*	=518,
cmLineEnd	=505,	*cmInsMode*	=519,
cmLineUp	=506,	*cmStartSelect*	=520,
cmLineDown	=507,	*cmHideSelect*	=521,
cmPageUp	=508,	*cmIndentMode*	=522,
cmUpdateTitle	=523;		
TEditor ダイアログ定数			
edOutOfMemory	=0,	*edSaveAs*	=6,
edOutOfMemory	=0,	*edSaveAs*	=6,
edReadError	=1,	*edFind*	=7,
edWriteError	=2,	*edSearchFailed*	=8,
edCreateError	=3,	*edReplace*	=9,
edSaveModify	=4,	*edReplacePrompt*	=10;
edSaveUntitled	=5,		
TEditor エディタフラグ			
efCaseSensitive	=0x0001,	*efReplaceAll*	=0x0008,
efWholeWordsOnly	=0x0002,	*efDoReplace*	=0x0010,
efPromptOnReplace	=0x0004,	*efBackupFiles*	=0x0100;
エディタパレット			
cpIndicator	"¥x02¥x03"		
cpEditor	"¥x06¥x07"		
cpMemo	"¥x1A¥x1B"		

データメンバ

autoIndent Boolean autoIndent ;

エディタが自動インデントモードにある場合は *True* です。

buffer char *buffer ;

テキストを保持するためのバッファへのポインタ。

bufLen ushort bufLen ;

バッファの最初から現在のカーソル位置までの間に保存されているテキストの量。582 ページを参照してください。

bufSize ushort bufSize ;

バッファのサイズ。

canUndo Boolean canUndo ;

エディタが ***undo*** をサポートする場合は *True* です。サポートしない場合は *False* です。

関連項目 ***TEditor::undo***

clipboard static TEditor *clipboard ;

クリップボードへのポインタ。***TEditor*** オブジェクトは，この変数に代入するだけでクリップボードにすることができます。クリップボードでは，アンドゥをサポートしないようにする必要があります（つまり，クリップボードの ***canUndo*** は *False* にしておく必要があります）。

curPos TPoint curPos ;

カーソルのファイル内での行位置および桁位置。

curPtr ushort curPtr ;

カーソルのオフセット。

delCount　ushort delCount ;

テキストから削除され，ギャップの最後に格納されている文字のバイト数。アンドゥを実現するために使用されます。

delta　TPoint delta ;

ビューとして表示されている最上行および最左端の桁。

drawLine　int drawLine ;

挿入された文字が表示される画面上の桁位置。***draw*** が内部的に使用します。

drawPtr　ushort drawPtr ;

現在のカーソルに対応するバッファのオフセット。***draw*** が内部的に使用します。

editorDialog　static TEditorDialog editorDialog ;

TEditorDialog データ型は int 引数を 1 個とり，その他に可変個の引数をとって，***ushort*** を返す関数へのポインタです。これは EDITORS.H で，次のように定義されています。

```
typedef ushort (*TEditorDialog)( int, ... );
```

editorDialog は，***TEditor*** オブジェクトがいろいろなダイアログボックスを表示するために使う関数ポインタです。ダイアログボックスはアプリケーションに応じてかなり異なるので，***TEditor*** オブジェクトは，それ自身のダイアログボックスを直接表示するのではなく，この関数ポインタを用いてそれらダイアログボックスの制御を行ないます。***editorDialog*** 関数の例については，TVEDIT.CPP を参照してください。

最初の int 引数で渡されるダイアログの値は，その名前が示すとおりの意味です。

edOutOfMemory	***edSaveAs***
edReadError	***edFind***
edWriteError	***edSearchFailed***

edCreateError	***edReplace***
edSaveModify	***edReplacePrompt***
edSaveUntitled	

デフォルトの ***editorDialog*** として定義されている ***defEditorDialog*** は，単に ***cmCancel*** を返します。

editorFlags static ushort editorFlags ;

editorFlags には，エディタで使ういろいろなフラグが入っています。

efCaseSensitive	大文字と小文字を区別して検索する。
efWholeWordsOnly	単語のみを対象として検索する。
efPromptOnReplace	置換の際に確認を行なう。
efReplaceAll	一括置換を行なう。
efDoReplace	置換を行なう。
efBackupFiles	セーブするときに.BAK ファイルを作成する。

デフォルト値は，***efBackupFiles*** | ***efPromptOnReplace*** です。

findStr static char findStr [80] ;

前回の検索に使用された文字列を保存します。

関連項目 ***TEditor::doSearchReplace***

gapLen ushort gapLen ;

カーソルの前にあるテキストとカーソルの後ろにあるテキストの「ギャップ」のサイズ。582 ページに説明されています。

hScrollBar TScrollBar *hScrollBar ;

水平スクロールバーへのポインタ。スクロールバーがないときは 0 です。

indicator TIndicator *indicator ;

インジケータへのポインタ。インジケータがないときは 0 です。

insCount ushort insCount ;

前回カーソルを移動してから，テキストに挿入された文字数。アンドゥを実現するために使います。

isValid Boolean isValid ;

ビューが有効なら *True* です。メンバ関数 ***valid*** が使用します。

関連項目 ***TEditor::valid***

keyState int keyState ;

CTRL-K などの特殊なキーが押されたことを示します。***handle-Event*** が，***CTRL-K-H*** や ***CTRL-K-B*** などの「二重の」コントロールキーを記録するために使用します。

limit TPoint limit ;

表示する桁の最大数，およびファイルの行数。スクロールバーの最大値を記録します。

lockCount uchar lockCount ;

ビューを再表示するタイミングを制御するためのロックカウントセマフォが入ります。***lockCount*** は ***lock*** によりインクリメントされ，***unlock*** によりデクリメントされます。

関連項目 ***TGroup::lock***, ***TGroup::unlock***, ***TGroup::lockFlag***

modified Boolean modified ;

バッファに変更があれば *True* です。

overwrite Boolean overwrite ;

上書きモードであれば *True* です。そうでなければ，エディタは挿入モードです。

replaceStr static char replaceStr [80] ;

前回の置換で使用された文字列を保存します。

関連項目　***TEditor::doSearchReplace***

selecting　Boolean selecting ;

エディタが選択モードにある（つまり，***CTRL-K CTRL-B*** が押された）なら *True* です。

selEnd　ushort selEnd ;

CTRL-K CTRL-K で選択されたテキストの最後を示すオフセット。584 ページを参照してください。

selStart　ushort selStart ;

CTRL-K CTRL-B で選択されたテキストの最初を示すオフセット。584 ページを参照してください。

updateFlags　uchar updateFlags ;

エディタの状態を示すフラグ群。***doUpdate*** などのメンバ関数は，これらのフラグを調べ，ビューを再表示する必要があるかどうかを判断します。

vScrollBar　TScrollBar *vScrollBar ;

水平スクローバーへのポインタ。スクロールバーがなければ 0 です。

メンバ関数

コンストラクタ　TEditor(const TRect& bounds, TScrollBar *aHScrollBar, TScrollBar *aVScrollBar, TIndicator * anIndicator, ushort aBufSize) ;

TView (*bounds*) を呼び出して，指定した境界を持つビューを作成します。***hScrollBar***，***vScrollBar***，***indicator***，および ***bufSize*** の各データメンバには，指定された引数が設定されます。スクロールバー，インジケータが必要ない場合には，それらの引数には 0 を設定します。デフォルト値の設定は以下のとおりです。

canUndo	*True*
selecting	*False*
overwrite	*False*
autoIndent	*False*
lockCount	0
keyState	0
growMode	***gfGrowHiX*** | ***gfGrowHiY***
options	***ofSelectable***
eventMask	***evMouseDown*** | ***evKeyDown*** | ***evCommand*** | ***evBroadcast***

次にバッファを割り当てて，クリアします。メモリサイズが不足する場合には，***editorDialog*** を使って ***edOutOfMemory*** ダイアログボックスを表示し，***isValid*** を *False* にします。充分なメモリサイズが得られた場合には，***isValid*** に *True* を設定します。エディタバッファに関連するデータメンバについても，当然ながら初期値設定を行ないます。***bufLen*** には 0，***gapLen*** には ***bufSize***，***selStart*** には 0，***modified*** には *False* などと設定していきます。

関連項目　***TView::Tview***, ***TEditor::editorDialog***, ***TScrollBar*** クラス

デストラクタ　virtual TEditor ( ) ;

エディタを消去し，バッファを削除します。

bufChar　char bufChar (ushort p) ;

ギャップを考慮した *p* 番目の文字（バイト）を返します。

bufPtr　ushort bufPtr (ushort p) ;

ギャップを考慮し，*p* 番目の文字（バイト）バッファにおけるオフセットを返します。

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***, ***ipstream::readData***

changeBounds　virtual void changeBounds (const TRect& bounds) ;

ビューの境界を変更し，***delta*** の値を再設定して，必要ならスクロールバーとビューの再表示を行ないます。親のサイズが変わったときにもファイルがビューの中に収まるようにオーバーライドされています。

charPos　int charPos (ushort p, ushort target) ;

バッファ内の *p* と *target* の間にある文字を調べて実際のカーソル位置を計算し，そのカーソル位置を返します。タブコードは，タブ設定のモジュロ個のスペースとしてカウントされます。

charPtr　ushort charPtr (ushort p, int target) ;

charPos の逆。カーソル位置に対応するバッファ内の位置を計算し，その位置を返します。

checkScrollBar　void checkScrollBar (const TEvent& event, TScrollBar * p, int& d) ;

handleEvent が，***cmScrollBarChanged*** ブロードキャストイベントを処理するときに呼び出してきます。スクロールバーの現在の ***value*** が *d* 以外なら，スクロールバーの再表示を行ないます。

clipCopy　Boolean clipCopy ( ) ;

このエディタがクリップボードを使用していない場合には，*False* を返します。クリップボードを使用している場合，次のようにして，選択テキストをエディタからクリップボードにコピーします。

(L1 TVCPP14－02)

選択テキストを選択解除し(ハイライトをやめる)，ビューを再表示します。

すべてうまくいった場合には *True* を返します。

関連項目　***TEditor::insertFrom***

clipCut void clipCut ( ) ;

clipCopy と同じですが，選択テキストをクリップボードにコピーした後で削除します。

関連項目 ***TEditor::clipCopy***，***TEditor::insertFrom***

clipPaste void clipPaste ( ) ;

clipCopy の逆。クリップボードの内容を，次のようにして，エディタの現在位置にコピーします（クリップボードを使用している場合)。

```
insertFrom( clipboard );
```

関連項目 ***TEditor::clipCopy***，***TEditor::insertFrom***

convertEvent virtual void convertEvent (TEvent& ev) ;

handleEvent がキーイベントをコマンドイベントに変換することで，基本的な編集処理を行なえるようにするために呼び出します。***convertEvent*** メンバ関数をオーバーライドし，これらデフォルトのキー割り当てを変更したり拡張することができます。584 ページを参照してください。

関連項目 ***TEditor::handleEvent***

cursorVisible Boolean cursorVisible ( ) ;

そのビューでカーソル（挿入点）の表示を行なっている場合は *True* を返します。

deleteRange void deleteRange (ushort startPtr, ushort endPtr, Boolean delSelect) ;

delSelect が *True* で，現在有効な選択テキストがある場合，その選択テキストを削除します。それ以外の場合，***startPtr*** から ***endPtr*** までの範囲を選択して削除します。

関連項目 ***TEditor::deleteSelect***

deleteSelect void deleteSelect ( ) ;

選択テキストがあれば，それを削除します。たとえば ***clipCopy*** を行なった後などに，選択ブロックの削除が行なわれます。

関連項目　***TEditor::deleteRange***

doneBuffer　virtual void doneBuffer ( ) ;

バッファを削除します。

doSearchReplace　void doSearchReplace ( ) ;

検索および検索・置換の両方で使用されます。どちらで使用されるかは，***editorFlags*** の状態とダイアログボックスに対するユーザの指定によります。***efDoReplace*** フラグが立っていない場合，***doSearchReplace*** は ***findStr*** の検索だけを行ない，置換は行ないません。***efDoReplace*** フラグが立っている場合には，この関数は ***findStr*** に一致する文字列を ***replaceStr*** で置換します。いずれの場合も，検索文字列が検出できなかったときには ***editorDialog(edSearchFailed)*** を呼び出します。***editorFlags*** の ***efPromptOnReplace*** フラグが立っている場合には，***edReplacePrompt*** ダイアログボックスが表示されます。この場合，置換を行なうかどうかは，そのときのユーザの指定によって決まります。***efReplaceAll*** フラグが立っている場合には，***cmCancel*** コマンドが検出されるまで，確認を求めることなく，一致したすべての文字列に対して置換を行なっていきます。

関連項目　***TEditor::findStr***, ***TEditor::replaceStr***, ***TEditor::editorDialog***, ***TEditor::find***, ***TEditor::replace***, ***TEditor::editorFlags***

doUpdate　void doUpdate ( ) ;

updateFlags が 0 の場合，何も起こりません。0 以外の場合，***updateFlags*** の各ビットの状態に応じて，ビューとそのスクロールバーを更新して再表示します。たとえば ***ufView*** が立っている場合，***drawView*** でビューの再表示を行ないます。またビューが ***sfActive*** の場合，***updateCommands*** でコマンドセットを更新します。更新を行なったら，***updateFlags*** を 0 にします。

draw　virtual void draw ( ) ;

TView::draw をオーバーライドし，エディタを表示します。

drawLines　void drawLines (int y, int count, ushort linePtr) ;

linePtr にある行を，行位置 *y* から *count* 回だけ表示します。

find　void find ( ) ;

現在設定されている ***findStr***，またはユーザが指定した文字列に一致する文字列を検索します。***find*** は，エディタダイアログを表示し，ユーザに新たに検索文字列を入力してもらうか，あるいはすでに設定されている ***findStr*** でよいことを確認してもらいます。新たに検索文字列が入力された場合，その文字列が新しい ***findStr*** となります（キャンセルしなかったとき）。***find*** はまず，以下のように定義されている ***TFindDialogRec*** オブジェクトを作成します。

```
struct TFindDialogRec
{
  TFindDialogRec( const char *str, ushort flgs )
  {
    strcpy( find, str );
    options = flgs;
  }
  char find[80];
  ushort options;
};
```

次に ***str*** と ***flags*** に，それぞれ現在の ***findStr*** と ***editorFlags*** を設定し，コンストラクタを呼び出します。そして，***edFind*** エディタダイアログで ***findStr*** を変更するか，そのままでよいことを確認するよう求めます。最後に，検索のみ・置換なし処理（***efDoReplace*** はオフ）として ***doSearchReplace*** を呼び出します。

関連項目　***TEditor::findStr***, ***TEditor::replaceStr***, ***TEditor::doSearchReplace***, ***TEditor::editorDialog***, ***TEditor::replace***, ***TEditor::editorFlags***

formatLine　void formatLine (void *buff, ushort linePtr, int x,
ushort color) ;

linePtr にある行を指定した *color* で書式化し，その結果を *buff* に入れます。***drawLines*** が使用します。

getMousePtr ushort getMousePtr (TPoint m) ;

画面上の点 *m* に対応する文字を指すバッファオフセットを返します。

関連項目 ***TEditor::charPtr***

getPalette virtual TPalette& getPalette ( ) const ;

"¥x06¥x07¥" として定義されている，デフォルトの ***TEditor*** パレット ***cpEditor*** を返します。エディタのパレットを変更したい場合はオーバーライドしてください。

handleEvent virtual void handleEvent (TEvent& ev) ;

エディタのイベント処理を行ないます。エディタで処理するコマンドを拡張したい場合には，オーバーライドしてください。デフォルトのハンドラは，まず ***TView::handleEvent(ev)*** を呼び出し，次に ***convertEvent*** を呼び出して，関連するすべての編集キーイベントをコマンドイベントに変換します。

hasSelection Boolean hasSelection ( ) ;

選択が行なわれている場合，つまり ***selStart*** が ***selEnd*** と等しくない場合，*True* を返します。これら 2 つのデータメンバが等しい場合，選択部分はなく *False* が返されます。

関連項目 ***TEditor::setSelect***

hideSelect void hideSelect ( ) ;

selecting を *False* にし，setSelect (curPtr, CurPtr, false) を呼び出して現在の選択部分の表示をやめます。

関連項目 ***TEditor::setSelect***，***TEditor::selecting***

initBuffer virtual void initBuffer ( ) ;

bufSize サイズのバッファを割り当て，***buffer*** にそのバッファへ

のポインタを設定します。

insertBuffer　Boolean insertBuffer (char　*p, ushort offset, ushort length, Boolean allowUndo, Boolean selectText);

最低位のテキスト挿入メンバ関数です。配列 *p* の（*p*［*offset*］から始まる）*length* バイト分のテキストを，バッファ（の **curPtr**）へ挿入します。*allowUndo* が *True* の場合，***insertBuffer*** はアンドゥー情報を記録します。*selectText* が *True* の場合，挿入テキストは選択されます。処理に成功したら，*True* を返します。失敗したら，そのことを示すダイアログボックスを起動し，*False* を返します。***insertBuffer*** は，***insertFrom*** と ***insertText*** から呼び出されます。アプリケーション側から直接呼び出す必要はまずありません。

関連項目　***TEditor::insertFrom***，***TEditor::insertText***

insertFrom　virtual Boolean insertFrom (TEditor　*editor);

引数 *editor* において現在ブロックマークが付けられている選択部分を，このエディタに挿入します。このメンバ関数を使って，***clipCut***，***clipCopy***，および ***clipPaste*** が実現されています。以下の定義を見れば，***insertFrom*** と ***insertBuffer*** の 2 つの関数の意味が理解しやすいと思います。

```
Boolean TEditor::insertFrom( TEditor *editor )
{
   return insertBuffer( editor->buffer,
                        editor->bufPtr(editor->selStart),
                        editor->selEnd - editor->selStart, canUndo,
                        isClipboard());
}
```

ここでは，引数 *allowUndo* にデータメンバ ***canUndo*** の値が設定されています。

また，このエディタがクリップボードを使用している場合には，引数 *selectText* は *True* になります。

関連項目　***TEditor::insertBuffer***，***TEditor::isClipboard***

insertText　Boolean insertText (void *text, ushort length, Boolean selectText);

指定した *text* からこのオブジェクトのバッファに *length* バイトをコピーします。*selectText* が *True* の場合，挿入したテキストは選択されます。この関数は，***insertBuffer*** を簡素化したものです。

関連項目　***TEditor::insertBuffer***

isClipboard　Boolean isClipboard ( ) ;

このエディタがクリップボードを使用している場合，*True* を返します。クリップボードを持たない場合は *False* を返します。

関連項目　***TEditor::clipboard***

lineEnd　ushort lineEnd (ushort p) ;

指定されたポインタ *p* を含んでいる行の最後を指すバッファオフセットを返します。

関連項目　***TEditor::lineStart***

lineMove　ushort lineMove (ushort p, int count) ;

count の符号に応じて，オフセット *p* を含んでいる行を *count* 行だけ上または下に移動します。

関連項目　***TEditor::prevLine***，***TEditor::nextLine***

lineStart　ushort lineStart (ushort p) ;

指定されたポインタ *p* を含んでいる行の先頭を指すバッファオフセットを返します。

関連項目　***TEditor::lineEnd***

lock　void lock ( ) ;

セマフォ ***lockCount*** をインクリメントします。

関連項目　***TEditor::lockCount***，***TEditor::unlock***

newLine　void newLine ( ) ;

現在のポインタのところにニューライン(復帰および改行)を挿入し

ます。***autoIndent*** が設定されている場合には，新しい行の先頭へのタブの挿入(必要な個数分)も行ないます。

関連項目　***TEditor::autoIndent***

nextChar　ushort nextChar (ushort p) ;

指定されたオフセット *p* のところにある文字の，次の文字を示すバッファオフセットを返します。

nextLine　ushort nextLine (ushort p) ;

指定されたオフセット *p* を含んでいる行の次の行の先頭を示すバッファオフセットを返します。

nextWord　ushort nextWord (ushort p) ;

指定されたオフセット *p* を含んでいるワードの，次のワードの先頭を示すバッファオフセットを返します。

prevChar　ushort prevChar (ushort p) ;

指定されたオフセット *p* のところにある文字の，１つ手前の文字を示すバッファオフセットを返します。

prevLine　ushort prevLine (ushort) ;

指定されたオフセット *p* を含んでいる行の，１つ手前の行の先頭を示すバッファオフセットを返します。

prevWord　ushort prevWord (ushort) ;

現在のカーソルがあるワードの，１つ手前にあるワードの先頭を示すバッファオフセットを返します。

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* から，エディタオブジェクトを読み出します。

関連項目　***TStreamableClass***，***ipstream***

replace　void replace ( ) ;

現在設定してある ***findStr*** に一致する文字列（またはユーザが新

たに入力した検索文字列)を，現在設定してある ***replaceStr***(またはユーザが新たに入力した置換文字列）で置換します。***replace*** は，検索文字列および置換文字列の両方を入力してもらうか，指定済みの ***findStr*** と ***replaceStr*** でよいことを確認してもらうために，エディタダイアログを表示します。新たに文字列を入力した場合，それらの文字列が新しい ***findStr*** および ***replaceStr*** となります(キャンセルしなかった場合)。***replace*** はまず，次のように定義されている ***TReplaceDialogRec*** オブジェクトを作成します。

```
struct TReplaceDialogRec
{
  TReplaceDialogRec( const char *str, const char *rep, ushort flgs )
  {
    strcpy( find, str );
    strcpy( replace, rep );
    options = flgs;
  }
  char find[80];
  char replace[80];
  ushort options;
};
```

次に *str* と *rep* に，それぞれ現在の ***findStr*** と ***replaceStr*** を入れ，また *flg* に現在の ***editorFlags*** を設定してコンストラクタを呼び出します。そして，***edReplace*** エディタダイアログで2つの文字列を変更するか，そのままでよいことを確認するよう求めます。最後に，検索・置換処理(***efDoReplace*** はオン)として，***doSearchReplace*** を呼び出します。

関連項目 ***TEditor::findStr***，***TEditor::replaceStr***，***TEditor::doSearchReplace***，***TEditor::editorDialog***，***TEditor::find***，***TEditor::editorFlags***

scrollTo void scrollTo (int x, int y) ;

y 行目の *x* 桁目を，エディタの左上隅に移動させます。

search Boolean search (const char *findStr, ushort opts) ;

オプションを指定し，エディタバッファ(***CurPtr*** から始まる)内の指定文字列を検索します。オプションは，以下のとおりです。

efCaseSensitive	大文字と小文字を区別して検索する

TE Editor

efWholeWordsOnly 単語のみを対象として検索する

一致する文字列が見つかった場合には *True* を返し，見つからなかった場合には *False* を返します。一致する文字列が見つかった場合には，そのテキストを選択します。

setBufLen void setBufLen (ushort length) ;

bufLen に *length* を設定し，その長さに従って ***gapLen*** と ***limit*** を再設定します。***selStart***, ***selEnd***, ***curPtr***, ***delta.x***, ***delta.y***, ***drawLine***, ***drawPtr***, ***delCount***, および ***insCount*** には，すべて 0 を設定します。***curPos*** には ***delta*** を設定し，***modified*** は *False* にします。ビューを更新し，必要なら再表示します。***TEditor*** コンストラクタが ***setBufLen***(0) を呼び出します。***insertBuffer*** も，***setBufLen*** を使用します。

関連項目 ***TEditor::update***, ***TEditor::TEditor***, ***TEditor*** の各データメンバ，***TEditor::insertBuffer***。582ページを参照してください。

setBufSize virtual Boolean setBufSize (ushort newSize) ;

バッファサイズを *newSize* に変更する前に，この関数を呼び出す必要があります。この関数は，バッファをこの新しいサイズにすることができる場合に *True* を返すようにしなければなりません。デフォルトでは，*newSize* が ***bufSize*** 以下の場合に *True* を返します。

setCmdState void setCmdState (ushort command, Boolean enable) ;

enable が *True* であるか *False* であるかによって，またエディタが ***sfActive*** であるかどうかによって，指定されたコマンドを有効にしたり無効にしたりします。エディタが選択ビューでない場合には，コマンドは常に無効にされます。***enableCommands*** および ***disableCommands*** のかわりに使用することのできる便利な関数です。

setCurPtr void setCurPtr (ushort p, uchar selectMode) ;

setSelect を呼び出し，***curPtr*** をオフセット *p* に変更します。***curPtr*** が選択テキストの先頭または最後にある場合には，*select-*

Mode の値によっては，さらに設定の変更が必要となります。

関連項目 ***TEditor::setSelect***，***TEditor::curPtr***

setSelect void setSelect (ushort newStart, ushort newEnd,
Boolean curStart) ;

指定された2つのファイル内オフセットで選択部分を設定し，必要ならビューの再表示を行ないます。このメンバ関数は，*curStart* の値（*True* または *False*）にしたがって，選択部分の前または後ろにカーソルを置きます。

setState virtual void setState (ushort aState, Boolean enable) ;

TView::setState をオーバーライドして，インジケータとスクロールバーを隠したり表示したりします。***setState*** はまず，***TView::setState*** を呼び出し，コマンドを有効または無効にします。これ以外のコマンドも有効または無効にしたい場合，***TView::setState*** でなく，***updateCommands*** をオーバーライドしてください。コマンドの状態を更新する必要がある場合には，***setState*** が必ず呼び出されます。

shutDown virtual void shutDown ( ) ;

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するために内部的に使用します。***shutDown*** は，***destroy*** が呼び出されたときに，関連するデータメンバの設定を適切に行なえるように，多くのクラスでオーバーライドされます。

関連項目 第6章**『信頼性の高いプログラムを開発するには』**

startSelect void startSelect ( ) ;

handleEvent が，**CTRL-K CTRL-B** による選択を検出したときに呼び出します。それまでの選択部分の表示を止め，***selecting*** を *True* にします。

関連項目 ***TEditor::handleEvent***，***TEditor:hideSelect***

toggleInsMode void toggleInsMode ( ) ;

overwrite データメンバを *True* から *False* に，あるいは *False* から *True* に反転します。***setState*** を呼び出して，カーソルの形状を変更します。

関連項目　***TEditor::overwrite***，***sfCursorIns***

trackCursor　void trackCursor (Boolean center) ;

カーソルを表示させます。*center* が *True* の場合，カーソルが *y*（行）方向に関して画面の中央にくるようにします。*x*（桁）の位置は変えません。

undo　void undo ( ) ;

前回のカーソル移動後の変更をアンドゥします。

unlock　void unlock ( ) ;

データメンバ ***lockCount*** が 0 でなければ，デクリメントします（***lockCount*** が 0 になると，***doUpdate*** が実行されます)。ロック/アンロックの機構を設けることにより，不必要にビューの再表示を行なわないようにします。

関連項目　***TEditor:doUpdate***，***TEditor::lock***，***TEditor:lockCount***

update　void update (uchar aFlags) ;

updateFlags データメンバに *aFlags* をセットします。***lockCount*** が 0 なら，***doUpdate*** を呼び出します。

関連項目　***TEditor:doUpdate***，***TEditor::lock***，***TEditor:lockCount***，***TEditor::updateFlags***

updateCommands　void updateCommands ( ) ;

コマンドを更新する必要があるときに，必ず呼び出されます。***cmUndo***, ***cmClip***, ***cmCopy*** などのコマンドを有効および無効にするときに使用します。

valid　virtual Boolean valid (ushort command) ;

指定された *command* がそのビューで有効であるかどうかを返します。デフォルトでは，***valid*** は ***isValid*** の値（***buffer*** が 0 でなければ *True*）を返します。

write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

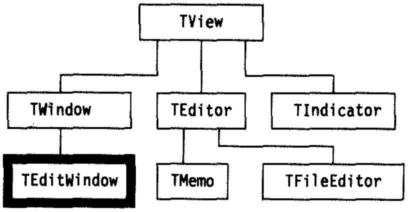
関連項目 ***TStreamableClass***，***opstream***

フレンド

関数 ***genRefs*** は ***TEditor*** のフレンド関数です。

関連する関数

演算子関数で ***TEditor*** に関係するものもありますが，メンバ関数とはなっていません。詳細については，第13章の 322 ページを参照してください。



TEditWindow は，***TFileEditor*** またはクリップボードを入れるためのウィンドウです。***TEditWindow*** では，タイトルを編集するファイル名が表示されるように変更し，エディタ用のスクロールバーとインジケータを初期化します。***TEditWindow*** に渡される名前がブランクであった場合，クリップボードの初期化を行なうものとみなされます。

データメンバ

editor TFileEditor *editor ;

このウィンドウにリンクされるエディタへのポインタ。

メンバ関数

コンストラクタ TEditWindow (const TRect& bounds, const char *fileName, int aNumber);

指定されたファイル名のファイルを編集する，ウィンドウ番号 *aNumber* の ***TEditWindow*** オブジェクトを作成します。スクロールバーとインジケータを備える，フレーム付きのタイル表示可能なウィンドウを初期化します。*fileName* が 0 なら，タイトルなしの

ファイルとして処理します。***editor*** が ***clipboard*** に等しい場合には，エディタはクリップボードであるとみなします。

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***，***ipstream::readData***

close virtual void close ( ) ;

TWindow::close をオーバーライドし，エディタがクリップボードである場合にウィンドウをクローズするかわりに表示しないようにします。

関連項目 ***TWindow::close***

getTitle virtual const char *getTitle (int) ;

編集されるファイルの名前を返します。エディタがクリップボードである場合には，"Clipboard" を返します。

handleEvent virtual void handleEvent (TEvent&) ;

cmUpdateTitle を獲得したときに，ウィンドウのフレームを再表示します。***TFileEditor::saveAs*** が編集中のファイルの名前に変更がある場合，ウィンドウのタイトルを変更するためにこの関数を呼び出します。

関連項目 ***TFileEditor::SaveAs***

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***，***ipstream***

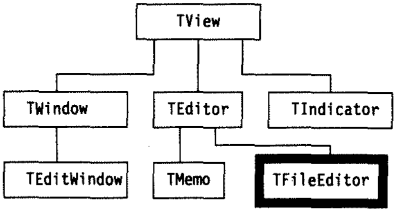
write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目 ***TStreamableClass***，***opstream***

関連する関数

演算子関数で ***TEditWindow*** に関係するものもありますが，メンバ関数とはなっていません。詳細については第13章の 322 ページを参照してください。



TFileEditor は，ファイルの内容を編集するために特殊化された ***TEditor*** の導出クラスです。

データメンバ

fileName char fileName [MAXPATH] ;

編集されるファイルの名前。

メンバ関数

コンストラクタ TFileEditor (const TRect& bounds, TScrollBar *aHScrollBar, TScrollBar * aVScrollBar, TIndicator *anIndicator, const char *aFileName) ;

指定されたスクロールバーとインジケータを備える ***TFileEditor*** オブジェクトを作成し，指定されたファイルの内容をロードします。ファイルが見つからなかった場合，またはファイルが無効な場合，エラーメッセージを表示しメンバ関数 ***valid*** は *False* を返します。***options*** 変数には ***sfSelectable*** を設定し，***eventMask*** にはブロードキャストイベントを処理するように設定します。***aHScrollBar***, ***aVScrollBar***，または ***anIndicator*** の各引数には，不要な場合には 0

をセットします。

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***，***ipstream::readData***

doneBuffer virtual void doneBuffer ( ) ;

バッファを消去します。

handleEvent virtual void handleEvent (TEvent& event) ;

TEditor::handleEvent を呼び出した後，***cmSave*** コマンドと ***cmSaveAs*** コマンドの処理を行ないます。***cmSave*** コマンドに対しては ***save*** を呼び出し，***cmSaveAs*** コマンドに対しては ***saveAs*** を呼び出します。

関連項目 ***TFileEditor::save***，***TFileEditor::saveAs***

initBuffer virtual void initBuffer ( ) ;

ファイルエディタバッファ用に ***bufSize*** バイトのメモリを割り当てます。

loadFile Boolean loadFile ( ) ;

ディスクから ***fileName*** ファイルを読み込み，エラーがないかどうかをチェックします。1つもエラーがない場合は *True*，エラーがある場合は *False* を返します。エラーの原因に応じて，***edOutOfMemory*** または ***edReadError*** ダイアログボックスを表示します。

関連項目 ***TEditor::editorDialog***

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***，***ipstream***

save Boolean save ( ) ;

編集中のファイルが "Untitled"の場合(つまり, ***fileName*** が設定されていない場合), ***saveAs*** を呼び出し, ***saveAs*** からの戻り値を返します。有効な ***fileName*** が設定されている場合, ***saveFile*** を呼び出し ***saveFile*** からの戻り値を返します。

関連項目 ***TFileEditor::saveAs***, ***TFileEditor::saveFile***

saveAs Boolean saveAs ( ) ;

セーブするファイル名を入力させるために, ***edSaveAs*** ダイアログを呼び出します。有効なファイル名を入力すると, 現在のテキストを ***saveFile*** メンバ関数を使ってこの名前でセーブします。そして, ***cmUpdateTitle*** ブロードキャストメッセージを発行して, このイベントをファイルエディタのオーナーに通知します。***saveFile*** に成功すると, ***saveAs*** は *True* を返します。成功しなかった場合には *False* を返されます。***edSaveAs*** ダイアログがキャンセルされた場合にも, *False* を返します。

関連項目 ***TFileEditor::saveFile***, ***TEditor::editorDialog***

saveFile Boolean saveFile ( ) ;

fileName ファイルをディスクにセーブします。セーブが失敗した場合は *False* を返し, 成功した場合は *True* を返します。***editorFlags*** の ***efBackupFiles*** ビットが立っている場合, .BAK ファイルを作成します。失敗した場合には, ***edCreateError*** または ***edWriteError*** ダイアログボックスを表示し, 失敗の原因を表示します。

関連項目 ***TEditor::editorDialog***, ***TFileEditor::saveAs***, ***TFileEditor::save***

setBufSize virtual Boolean setBufSize (ushort newSize) ;

TEditor::setBufSize をオーバーライドし, バッファを拡張または縮小します。バッファの拡張および縮小は 4K バイト単位で行ないます。またそれに呼応して, ***gapLen*** の再設定も行ないます。

updateCommands virtual void updateCommands ( ) ;

TEditor::updateCommands を呼び出した後，***cmSave*** コマンドと ***cmSaveAs*** コマンドを有効にします。これらのコマンドは，エディタが選択ビューとなっている場合のみ有効で，それ以外の場合無効にしておく必要があります。

関連項目 ***TEditor:updateCommands***

valid virtual Boolean valid (ushort) ;

TEditor::valid をオーバーライドしてプログラムを終了する前にファイルをセーブしないでよいのかを確認するための警告を行ないます。***edSaveUntitled*** か ***edSaveModify*** の，どちらか適当なダイアログを表示します。ユーザがセーブをキャンセルした場合，*False* を返します。

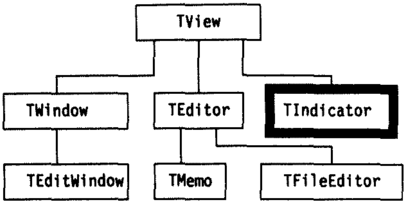
write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目 ***TStreamableClass***，***opstream***

関連する関数

演算子関数で ***TFileEditor*** に関係するものもありますが，メンバ関数とはなっていません。詳細については第13章の 322 ページを参照してください。



TIndicator は，編集ウィンドウの左下隅にある行と桁のカウンタです。***TEditWindow*** コンストラクタによって初期化され，***TEditor*** コンストラクタに 4 番目の引数として渡されます。

データメンバ

location TPoint location ;

表示位置をセーブします。***TEditor*** によって更新されます。

modified Boolean modified ;

リンクしている ***TEditor*** に変更があった場合に *True* にされます。

関連項目 ***TIndicator::draw***

メンバ関数

コンストラクタ TIndicator (const TRect& bounds) ;

TIndicator オブジェクトを作成します。

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するため

に呼び出されます。

関連項目　***TStreamableClass***, ***ipstream::readData***

draw　virtual void draw ( ) ;

インジケータを表示します。***modified*** が *True* の場合，アスタリスク（*）を表示します。

getPalette　virtual TPalette& getPalette ( ) const ;

TIndicator デフォルトパレット ***cpIndicator***（="¥x02¥x03"）を返します。

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***, ***ipstream***

setState　virtual void setState (ushort aState, Boolean enable) ;

ビューがドラッグされている場合，フレームドラッグ用の色でインジケータを表示します。

setValue　void setValue (const TPoint& aLocation, Boolean aModified) ;

リンクしている ***TIndicator*** オブジェクトのデータメンバの値を，更新および表示するために ***TEditor*** が呼び出します。

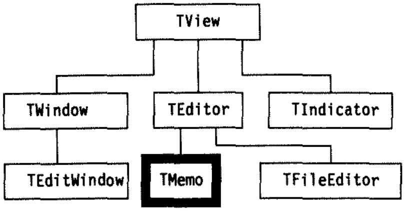
write　virtual void write (opstream& os) ;

出力ストリーム os への書き込みを行ないます。

関連項目　***TStreamableClass***, ***opstream***

関連する関数

演算子関数で ***TIndicator*** に関係するものもありますが，メンバ関数とはなっていません。詳細については第13章の 322 ページを参照してください。



TEditor の導出クラスである ***TMemo*** は，ダイアログや入力フォームなどにテキストの入力を行なうためのクラスです。***dataSize*** をサポートし，***TDioalog*** で ***TAB*** キーを処理できるようにします。***TEditor*** のパレットとは別のパレットをもっています。***dataSize*** は，以下の構造体を使用し，サイズとして (bufSize + sizeof (ushort)) を返します。

```
struct TMemoRec
{
   ushort length;
   char *buffer;
}
```


メンバ関数

コンストラクタ　TMemo ( ) ;

ベースのコンストラクタを利用して，***TMemo*** のオブジェクトを作成します。

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

dataSize virtual ushort dataSize ( ) ;

TMemo オブジェクトをセーブまたは読み出したりする際に，***TView*** から継承された ***setData*** および ***getData*** と一緒に使用されます。デフォルト処理として，(sizeOf (ushort) + bufSize) を返します。

関連項目 ***TView::getData***, ***TView::setData***

getPalette virtual TPalette& getPalette ( ) const ;

デフォルトのメモパレット ***cpMemo*** (="¥x1A¥x1B") を返します。

handleEvent virtual void handleEvent (TEvent& event) ;

TMemo が ***kbTab*** キーイベントを処理できないようにします。それ以外の場合，***TEditor*** と同様にイベントを処理します。

関連項目 ***TEditor::handleEvent***

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***, ***ipstream***

write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目 ***TStreamableClass***, ***opstream***

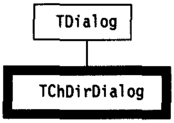
関連する関数

演算子関数で ***TMemo*** に関係するものもありますが，メンバ関数とはなっていません。詳細については第 13 章の 322 ページを参照してください。

第 15 章
標準ダイアログボックスの実現

この章では，標準的なダイアログボックスにいろいろな機能を付加するための Turbo Vision の拡張機能について解説します。

TChDirDialog　　STDDLG.H



TChDirDialog は，Change Directory というラベルのついたダイアログボックスです。ユーザがディレクトリ名を指定できるように入力行が用意されています。また，それまでに選択してきたディレクトリや，現在アクセスすることのできる全ディレクトリのツリーを表示するための垂直スクロールバー付きのヒストリーウィンドウとディレクトリリストボックスも用意されています。マウスとキーボードによる選択も，ハイライトさせてクリックするという通常のやり方で行なうことができます。以下のものを表示させることができます。

Directory name（ディレクトリ名）
Directory tree（ディレクトリツリー）
OK（デフォルト）
Chdir
Revert
Help

TChDirDialog:handleEvent は，これら選択項目に対応する所定のコマンドを生成します。

TDirListBox は ***TChDirDialog*** のフレンドなので，***TDirListBox*** のメンバ関数は ***TchDirDialog*** のプライベートメンバ，すなわち ***setUpDialog***，***dirInput***，***dirList***，***okButton***，および ***chDirButton*** にアクセスすることができます。

コマンド定数

TChDirDialog が生成するコマンドは以下のとおりです。

表 15.1 TChDirDialog のコマンド

定数	値	意味
cmChangeDir	1005	Chdir を選択したときに生成されます。
cmRevert	1006	Revert を選択したときに生成されます。

メンバ関数

コンストラクタ　TChDirDialog (ushort aOptions, ushort histId) ;

指定したヒストリー ID を持つディレクトリ変更用ダイアログオブジェクトを作成します。引数 *aOptions* では，以下のような設定を行ないます。

- ***cdNormal***：すぐにダイアログを使うためのオプション。
- ***cdNoLoadDir***：カレントディレクトリをロードせずに，ダイアログを初期化するオプション。***setData*** を使ってディレクトリをリセットしたい場合，またはストリームへのセーブを行なう前に使います。
- ***cdHelpButton***：ダイアログにヘルプボタンを入れるためのオプション。

コンストラクタは，***TInputLine*** のオブジェクト（Directory~n~ame というラベルが付く），***THistory*** のオブジェクト，垂直スクロールバー，***TDirListBox*** オブジェクト（Directory~t~ree というラベルが付く），およびボタン 3 個（O~K~, ~C~hdir, ~R~evert）を作成しインサートします。*aOptions* の ***cdHelpButton*** フラグが立っている場合，4 個目のボタンとしてヘルプボタンが作成されます。***cdNoLoadDir*** オプションが立っていなければ，ダイアログボックスにはカレントディレクトリがロードされます。

build static TStreamable *build( );

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***, ***ipstream::readData***

dataSize virtual ushort dataSize( );

dataSize は，デフォルト処理として 0 を返します。オーバーライドして，***getData*** および ***setData*** がダイアログボックスの入力データを保存したり読み出すときに使用するデータサイズ（バイト単位）を返すようにしてください。

関連項目 ***TView::dataSize***, ***TView::setData***, ***TView::getData***

getData virtual void getData (void *rec) ;

getData は，デフォルトでは何も行ないません。オーバーライドし，ビューから *rec* に ***dataSize*** バイト分コピーするようにしてください。ダイアログボックスの入力データを保存したり読み出したりする目的で，***dataSize*** および ***setData*** と一緒に使用されます。

関連項目 ***TView::dataSize***, ***TView::setData***, ***TView::getData***

handleEvent virtual void handleEvent (TEvent& event) ;

TDialog::handleEvent を呼び出してから，***cmRevert*** イベント（前回のカレントディレクトリの復元）および ***cmChangeDir*** イベント（選択したディレクトリへの切り換え）の処理を行ないます。ダイアログは，必要に応じて再表示されます。

関連項目　***TDialog::handleEvent***

read　virtual void *read (ipstream& is) ;

入力ストリーム ***is*** からの読み出しを行ないます。

関連項目　***TStreamableClass***，***ipstream***

setData　virtual void setData (void *rec) ;

setData は，デフォルトで何も行ないません。オーバーライドして，*rec* からビューに ***dataSize*** バイト分コピーするようにしてください。ダイアログボックスの入力データを保存したり読み出す目的で，***dataSize*** および ***getData*** と一緒に使用されます。

関連項目　***TView::dataSize***，***TView::setData***，***TView::getData***

shutDown　virtual void shutDown ( ) ;

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するために，内部的に使用します。***shutDown*** は，***destroy*** が呼び出されたときに，関連するデータメンバの設定を適切に行なえるようにするために多くのクラスでオーバーライドされます。

関連項目　第6章**『信頼性の高いプログラムを開発するには』**

valid　virtual Boolean valid (ushort command) ;

command が ***cmOK*** 以外である場合に *True* を返します。OK ボタンが押された場合，***valid*** は入力行をチェックし，有効なディレクトリであれば，*True* を返します。無効なディレクトリだった場合には，「無効ディレクトリ」のメッセージボックスを呼び出し *False* を返します。

関連項目　***TDialog::valid***

write　virtual void write (opstream& os) ;

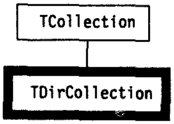
出力ストリーム os への書き込みを行ないます。

関連項目　***TStreamableClass***, ***opstream***

関連する関数

演算子関数で ***TChDirDialog*** に関係するものもありますが，メンバ関数とはなっていません。詳細については第 13 章の 322 ページを参照してください。

TDirCollection　　STDDLG.H



TDirCollection は ***TCollection*** の単純な導出クラスであり，***TDirEntry*** オブジェクトを保存するために使用されます。***TDir-Collection*** はストリーム入出力可能なクラスで，その基本クラスから ***TStreamable*** を継承します。。

メンバ関数

コンストラクタ　TDirCollection (ccIndex aLimit, ccIndex aDelta):
　　TCollection(aLimit, aDelta);

ベースの ***TCollection*** コンストラクタを呼び出して，指定された ***limit*** と ***delta*** を持つディレクトリコレクションを作成します。

関連項目 ***TCollection::TCollection***

at TDirEntry *at (ccIndex index) ;

このディレクトリコレクション内で，*index* によってインデックスされている ***TDirEntry*** オブジェクトへのポインタを返します。

関連項目 ***TNSCollection::at***

atInsert void atInsert (ccIndex index, TDirEntry *item) ;

指定の *item* をコレクションの *index* で指定される位置に挿入し，それ以降の要素を 1 つずつ後ろにずらしていきます。その挿入によって ***limit*** を越えた場合，コレクションは ***delta*** 個分拡張されます。

関連項目 ***TNSCollection::atInsert***

atPut void atPut (ccIndex index, TDirEntry *item) ;

index の位置にある要素を，指定の *item* で置き換えます。

関連項目 ***TNSCollection::atPut***

build static TStreamable *build ( );

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***, ***ipstream::readData***

firstThat TDirEntry *firstThat (ccTestFunc Test, void *arg) ;

この繰り返し処理関数は，コレクション内で ***Test*** 関数に対して最初に *True* を返してきた ***TDirEntry*** オブジェクトへのポインタを返します。詳細については，***TNSCollection::firstThat*** を参照してください。

関連項目 ***TDirCollection::lastThat***

free void free (TDirEntry *item) ;

コレクションから指定の要素を削除(消去)し，コレクション内の領

域を解放します。

関連項目 ***TNSCollection::free, TNSCollection::remove***

indexOf virtual ccIndex indexOf (TDirEntry *item) ;

指定された *item* の，このディレクトリコレクション内における ***index*** を返します。

関連項目 ***TNSCollection::indexOf***

insert virtual ccIndex insert (TDirEntry *item) ;

コレクションに *item* を挿入し，必要に応じて他のインデックスを調整します。デフォルトでは，コレクションの最後に挿入されます。挿入された要素のインデックスが返されます。

関連項目 ***TNSCollection::insert***

lastThat TDirEntry *lastThat (ccTestFunc Test, void *arg) ;

この繰り返し処理関数は，コレクションを最後の ***TDirEntry*** オブジェクトから始めて，最初の ***TDirEntry*** オブジェクトに向かって検索していきます。この関数は，コレクション内で ***Test*** 関数に対して最初に *True* を返してきた（つまり末尾にもっとも近い）要素へのポインタを返します。詳細については，***TNSCollection::lastThat*** を参照してください。

関連項目 ***TDirCollection::firstThat***

read virtual void *read (ipstream& is, void *t) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***, ***ipstream***

readItem void *TDirCollection::readItem (ipstream& is) ;

コレクション内の各要素に対して呼び出されます。要素を正しく読み込むには，***TCollection*** または ***TSortedCollection*** から導出されたすべてのクラスでこのメンバ関数をオーバーライドする必要があります。***TSortedCollection*** では，すでにこの関数をオーバーラ

TD
Dialogs

イドしています。

関連項目 ***TStreamableClass***, ***TStreamable***, ***ipstream***

remove void remove (TDirEntry *item) ;

このコレクションから指定された *item* を削除 (消去) します (コレクション内の領域は解放されません)。

関連項目 ***TNSCollection::remove***, ***TNSCollection::free***

write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目 ***TStreamableClass***, ***opstream***

writeItem void TDirCollection::writeItem (void *obj, opstream& os) ;

コレクション内の各要素に対して呼び出されます。要素を正しく書き込むには, ***TCollection*** または ***TSortedCollection*** から導出されたすべてのクラスでこのメンバ関数をオーバーライドする必要があります。***TSortedCollection*** では, すでにこの関数をオーバーライドしています。

関連項目 ***TStreamableClass***, ***TStreamable***, ***opstream***

関連する関数

演算子関数で ***TDirCollection*** に関係するものもありますが, メンバ関数とはなっていません。詳細については第 13 章の 322 ページを参照してください。

TDirEntry STDDLG.H

TDirEntry は，ディレクトリパスとテキストを格納する単純なクラスです。***TDirEntry*** オブジェクトは，***TDirCollection*** オブジェクトに保存されます。***TDirEntry*** にはプライベートの ***char**** データメンバが 2 個あり，メンバ関数 ***dir*** および ***text*** を介してアクセスすることができます。

メンバ関数

コンストラクタ TDirEntry(const char * txt, const char * dir);
inline TDirEntry::TDirEntry (const char * txt, const char * dir) :
displayText (newStr (txt)), directory (newStr (dir))

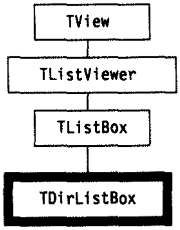
dir char *dir( );

カレントディレクトリ（プライベートデータメンバ ***directory*** の値）を返します。

text char *text( );

現在の表示テキスト（プライベートデータメンバ ***displayText*** の値）を返します。

TDirListBox STDDLG.H



TDirListBox は ***TListBox*** を特殊化した導出クラスであり，***TDirCollection*** オブジェクトに格納されているディレクトリを表示したり選択したりするためのものです。デフォルトではディレク

トリを 1 段にして表示し，垂直スクロールバーを持たせることができます。

メンバ関数

コンストラクタ　TDirListBox（const TRect& bounds, TScrollBar * aScrollBar）;

TListBox::TListBox(bounds, 1, aScrollBar) を呼び出し，指定された境界と垂直スクロールバーを備えた 1 段のリストボックスを作成します。

関連項目　***TListBox::TListBox***

デストラクタ　TDirListBox（）;

ベースのデストラクタを呼び出して，リストボックスを廃棄します。

build　static TStreamable *build（）;

ストリーム読み出しを行なう際，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

getText　virtual void getText（char *dest, short item, short maxLen）;

インデックス *item* の位置にあるテキスト文字列を取り出し，それを *dest* にコピーします。

handleEvent　virtual void handleEvent（TEvent& event）;

ダブルクリックのマウスイベントを受け取ると，putEvent(cmChangeDir) を呼び出します。したがって，ダブルクリックで選択したディレクトリに変更されます。その他のイベントは，***TListBox::handleEvent*** が処理します。

関連項目　***TView::putEvent***，***TListBox::handleEvent***

isSelected　virtual Boolean isSelected（short item）;

item が選択されているなら *True* を返し，選択されていなければ *False* を返します。

list TDirCollection *list ( ) ;

このディレクトリリストボックスに現在リンクされている ***TDir-Collection*** オブジェクトへのポインタを返します。

関連項目 ***TSortedListBox::list***

newDirectory void newDirectory (const char *) ;

それまでこのリストボックスにリンクされてきた ***TDirEntry*** オブジェクトを削除し，*aList* で指定されているファイルコレクションをそのかわりとします。新しいコレクションでは，最初の要素にフォーカスがきます。

関連項目 ***TSortedListBox::newList***

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***, ***ipstream***

setState virtual void setState (ushort aState, Boolean enable) ;

デフォルト処理として，先祖の ***TListViewer::setState*** を使います。

関連項目 ***TListViewer::setState***

write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

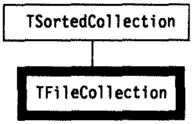
関連項目 ***TStreamableClass***, ***opstream***

関連する関数

演算子関数で ***TDirListBox*** に関係するものもありますが，メンバ

関数とはなっていません。詳細については第13章の322ページを参照してください。

TFileCollection　　STDDLG.H



TFileCollection は ***TSortedCollection*** の単純な導出クラスで，ファイル名を要素とするソート済みのコレクションです。ファイル名は，データ型 ***TSearchRec*** のオブジェクト（その定義は以下のとおり）として格納されます。

```
struct TSearchRec
{
    uchar attr;
    long time;
    long size;
    char name[MAXFILE+MAXEXT-1];
};
```

TSearchRec の各フィールドは，DOS における通常の意味と同じです。ドライブとディレクトリパスは，***TDirEntry*** オブジェクトを使って別個に処理されます。

メンバ関数

コンストラクタ　TFileCollection (ccIndex aLimit, ccIndex aDelta) :
　　TSortedCollection (aLimit, aDelta) {}

ベースの ***TSortedCollection*** コンストラクタを呼び出し，指定された ***limit*** と ***delta*** を持つコレクションを作成します。

関連項目　***TSortedCollection::TSortedCollection***

at　TSearchRec *at (ccIndex index)

このファイルコレクションにおいて，*index* でインデックスされる ***TSearchRec*** オブジェクトへのポインタを返します。

関連項目 ***TNSCollection::at***

atInsert void atInsert（ccIndex index, TSearchRec *item）

item で参照される ***TSearchRec*** ファイルを，コレクション内の *index* で指定される位置に挿入し，それ以降の要素を 1 つずつ後ろにずらしていきます。挿入によって ***limit*** を越える場合，コレクションは ***delta*** 個分拡張されます。

関連項目 ***TNSCollection::atInsert***

atPut void atPut（ccIndex index, TSearchRec *item）

index の位置に見つかった ***TSearchRec*** ファイルを，指定の *item* で置き換えます。

関連項目 ***TNSCollection::atPut***

build static TStreamable *build（）;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***，***ipstream::readData***

compare virtual int compare（void *key1, void *key2）; private

標準的なファイル文字列の比較を行ない，その結果に対応した値を返します。

■ *key1* および *key2* のファイル名が同じ場合，0 を返します。

■ 次の場合，> 0 を返します。
- *key1* の名前が，辞書順で *key2* の名前よりも上位（後ろ）の場合。
- *key1* の名前がディレクトリ .. の場合。
- *key1* がディレクトリで，*key2* がディレクトリでない場合。

■ 次の場合，< 0 を返します。

・ *key1* の名前が，辞書順で *key2* の名前よりも下位（前）の場合。
・ *key2* がディレクトリ .. の場合。
・ *key2* がディレクトリで，*key1* がディレクトリでない場合。

関連項目　***TSortedCollection::compare***

firstThat　TSearchRec *firstThat (ccTestFunc Test, void *arg) ;

この繰り返し処理関数 (iterator) は，コレクション内で ***Test*** 関数に対して最初に *True* を返してきた ***TSearchRec*** オブジェクトへのポインタを返します。詳細については，***TNSCollection::firstThat*** を参照してください。

関連項目　***TFileCollection::lastThat***

free　void free (TSearchRec *item)

指定された ***TSearchRec*** ファイル *item* をコレクションから削除(消去)して，コレクション内の領域を解放します。

関連項目　***TNSCollection::free. TNSCollection::remove***

indexOf　virtual ccIndex indexOf (TSearchRec *item)

指定された ***TSearchRec*** ファイル *item* の，このファイルコレクションでの ***index*** を返します。

関連項目　***TNSCollection::indexOf***

insert　virtual ccIndex insert (TSearchRec *item)

コレクションに ***TSearchRec*** の *item* を挿入してから，必要に応じてその他のインデックスを調整します。デフォルトでは，コレクションの末尾に挿入します。挿入した要素のインデックスを返します。

関連項目　***TNSCollection::insert***

lastThat　TSearchRec *lastThat (ccTestFunc Test, void *arg) ;

この繰り返し処理関数は，コレクションを最後の要素から始めて，

最初の要素に向かって検索していきます。この関数は，コレクション内で ***Test*** 関数に対して最初に *True* を返してきた（つまり末尾にもっとも近い）要素へのポインタを返します。詳細については，***TNSCollection::lastThat*** を参照してください。

関連項目　***TFileCollection::firstThat***

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***，***ipstream***

readItem　void *TFileCollection::readItem (ipstream& is) ;

コレクション内の各要素に対して呼び出されます。オーバーライドして，要素を正しく読み出すようにする必要があります。

関連項目　***TStreamableClass***，***TStreamable***，***ipstream***

remove　void remove (TSearchRec *item)

指定された ***TSearchRec*** ファイル *item* をコレクションから削除(消去)します（コレクション内の領域は解放されません)。

関連項目　***TNSCollection::remove***，***TNSCollection::free***

write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目　***TStreamableClass***，***opstream***

writeItem　void TFileCollection::writeItem (void *obj, opstream& os) ;

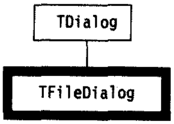
コレクション内の各要素に対して呼び出されます。この種の関数は，オーバーライドして要素を正しく書き込むようにする必要があります。

関連項目　***TStreamableClass***，***TStreamable***，***opstream***

関連する関数

演算子関数で ***TFileCollection*** に関係するものもありますが，メンバ関数とはなっていません。詳細については第 13 章の 322 ページを参照してください。

TFileDialog　　STDDLG.H



TFileDialog は，ファイルダイアログボックス，ヒストリー選択リスト，および入力行を作成し，ファイル名（ワイルドカードを含む）の入力，編集，選択，あるいはそのファイルをオープンして編集できるようにします。

オプション

以下のオプションで，動作の指定を行ないます。

表 15.2
TFileDialog の
オプション

定数	値	意味
fdOKButton	0x0001	ダイアログに OK ボタンを入れます。
fdOpenButton	0x0002	ダイアログに Open ボタンを入れます。
fdReplaceButton	0x0004	ダイアログに Replace ボタンを入れます。
fdClearButton	0x0008	ダイアログに Clear ボタンを入れます。
fdHelpButton	0x0010	ダイアログに Help ボタンを入れます。
fdNoLoadDir	0x0100	初期化の際に，ダイアログにカレントディレクトリの内容をロードしません。つまり，***setData*** を使ってワイルドカードを変更したいか，またはストリームにダイアログを保存したいということです。

コマンド定数

TFileDialog は，以下のコマンドを返します。

表 15.3
TFileDialog の
コマンド定数

定数	値	意味
cmFileOpen	1001	Open を押したときに返されます。
cmFileReplace	1002	Replace を押したときに返されます。
cmFileClear	1003	Clear を押したときに返されます。
cmFileInit	1004	***TFileDialog::valid*** が使用します。

データメンバ

directory const char *directory ;

カレントディレクトリ。

fileList　TFileList *fileList;

リンクしているファイルリストへのポインタ。

関連項目　***TFileList***

fileName　TFileInputLine *fileName;

リンクしている入力行へのポインタ。

関連項目　***TFileInput***

wildCard　char wildCard [MAXPATH];

現在のドライブ，パス，およびファイル名。

メンバ関数

コンストラクタ　TFileDialog(const char *aWildCard, const char *aTitle,
　　const char *inputName, ushort aOptions, uchar histId);

指定のタイトルを持ち，サイズが固定で，フレーム付きのダイアログボックスを作成します。***TFileInputLine*** オブジェクト（***fileName*** データメンバで参照される）を作成して挿入し，*inputName* というラベルをつけ，その ***data*** データメンバに *aWildCard* を設定します。***fileList*** データメンバで参照される ***TFileList*** オブジェクトを得るために，ワイルドカード引数を（必要に応じて）展開します。***THistory*** オブジェクトを，指定のヒストリー ID を付けて作成，挿入します。また，垂直スクロールバーを 1 個作成し挿入します。*aOptions* フラグに応じて，ボタンを作成します（***TFileDialog*** のオプションの節を参照してください）。***TFileInfoPlane*** オブジェクトを作成し挿入します。***fdNoLoadDir*** フラグが設定されてない場合には，カレントディレクトリのファイルをファイルリストにロードします。

関連項目　***TDialog::TDialog***，***TFileInputLine::TFileInput-***

Line, ***TFileList::TFileList***, ***THistory::THistory***

デストラクタ　TFileDialog ( ) ;

directory を削除してから，ファイルダイアログを消去します。

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***, ***ipstream::readData***

getFileName　void getFileName (char *s) ;

fileName－＞data フィールドを取り，それをフルパスの形式に直します。結果は *s* に入ります。

handleEvent　virtual void handleEvent (TEvent& event) ;

TDialog::handleEvent を呼び出した後，***cmFileOpen***, ***cmFileReplace***, および ***cmFileClear*** の各イベントを処理します。これらのイベントはすべて，***endModal*** を呼び出してそのファイルダイアログをオープンしたビューにコマンドを渡します。

関連項目　***TDialog::handleEvent***, ***TView::endModal***

read　virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***, ***ipstream***

shutDown　virtual void shutDown( );

TObject::destroy が，導出したオブジェクトや関連するオブジェクトを間違いなく消去するために内部的に使用します。***shutDown*** は，***destroy*** が呼び出されたときに，関連するデータメンバの設定を適切に行なえるように，多くのクラスでオーバーライドされます。

関連項目　第 6 章**『信頼性の高いプログラムを開発するには』**

valid　virtual Boolean valid (ushort command) ;

command が ***cmValid*** であれば，うまく作成できたという意味で *True* を返します。そうでない場合には，***TDialog::valid*** を呼び出します。これが *True* を返してきた場合，現在の ***fileName*** が有効かどうかチェックします。有効な名前であれば，*True* を返します。無効な名前だった場合には，「Invalid file name（無効なファイル名）」のメッセージボックスを呼び出し *False* を返します。

関連項目　***TDialog::valid***

write　virtual void write (opstream& os)；

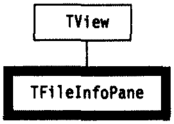
出力ストリーム *os* への書き込みを行ないます。

関連項目　***TStreamableClass***，***opstream***

関連する関数

演算子関数で ***TFileDialog*** に関係するものもありますが，メンバ関数とはなっていません。詳細については第 13 章の 322 ページを参照してください。

TFileInfoPane STDDLG.H



TFileInfoPane は，オーナーのファイルダイアログボックスにファイル情報を表示するための単純でストリーム入出力可能なビューです。

データメンバ

file_block TSearchRec file_block;

この情報ペインで使用するファイル名と属性。***TSearchRec*** の定義は次のとおりです。

```
struct TSearchRec
{
    uchar attr;
    long time;
    long size;
    char name[MAXFILE+MAXEXT-1];
};
```

フィールドは，通常の DOS ファイルの場合と同じです。

メンバ関数

コンストラクタ TFileInfoPane (const TRect& bounds) ;

TView::TView *(bounds)* を呼び出して，指定された *bounds* を持つファイル情報ペインを作成します。***eventMask*** に ***evBroadcast*** を設定します。

build static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目 ***TStreamableClass***，***ipstream::readData***

draw virtual void draw ( ) ;

デフォルトパレットを使用してファイル情報ペインを表示します。ブロックサイズと日時スタンプが表示されます。

getPalette virtual TPalette& getPalette ( ) const ;

デフォルトパレット ***cpInfoPane***（= ¥x1E）を返します。

handleEvent virtual void handleEvent (TEvent& event) ;

TView::handleEvent を呼び出した後，ブロードキャストイベント ***cmFileFocused***（ファイルリスト内の新規のファイルにフォーカスがきたときに発行される）に対して，ファイル情報ペインを表示します。

関連項目 ***TView::handleEvent***，***TFileList*** コマンド．

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***，***ipstream***

write virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

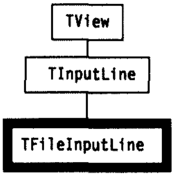
関連項目 ***TStreamableClass***，***opstream***

関連する関数

演算子関数で ***TFileInfoPane*** に関係するものもありますが，メンバ関数とはなっていません。詳細については第 13 章の322 ページを

参照してください。

TFileInputLine　STDDLG.H



TFileInputLine は ***TInputLine*** を特殊化したもので，ファイル名の入力と編集を行ないます。ファイル名には，パスやワイルドカードを含ませることもできます。***TFileInputLine*** オブジェクトは，ファイルダイアログボックスにリンクされます。

メンバ関数

コンストラクタ　TFileInputLine (const TRect& bounds, short aMaxLen) ;

TInputLine::TInputLine (bounds, aMaxLen) を呼び出して指定された境界と最大長を持つファイル入力行を作成します。***event-Mask*** に ***evBroadcast*** を設定します。

関連項目　***TInputLine::TInputLine***

build　static TStreamable *build ( ) ;

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***，***ipstream::readData***

handleEvent　virtual void handleEvent (TEvent& event) ;

TInputLine::handleEvent を呼び出した後，ブロードキャストイ

ベント ***cmFileFocused*** に対する処理として入力されたファイル名を入力行のデータメンバ ***data*** にコピーし，ビューを再表示します。編集した名前がディレクトリの場合，最初はオーナーの ***TFileDialog*** オブジェクトにおける現在のファイル名を付加します。

関連項目　***TInputLine::handleEvent***

read　virtual void *read (ipstream& os)；

入力ストリーム *is* からの読み出しを行ないます。

関連項目　***TStreamableClass***，***ipstream***

write　virtual void write (opstream& os)；

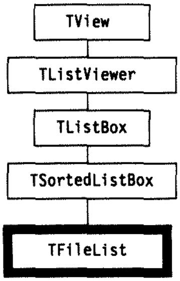
出力ストリーム *os* への書き込みを行ないます。

関連項目　***TStreamableClass***，***opstream***

関連する関数

演算子関数で ***TFileInputLine*** に関係するものもありますが，メンバ関数とはなっていません。詳細については第 13 章の322 ページを参照してください。

TFileList　　STDDLG.H



TFileList は，ファイル名(***TFileCollection*** オブジェクトに格納される）を要素とするソート済みの 2 段のリストボックスです。マウスまたはキーボードカーソルを使用し，ファイル名を選択する（ハイライトする）ことができます。垂直スクロールバーを付けることもできます。***TFileList*** は，ほとんどの機能を ***TSortedListBox*** から継承します。

コマンド定数

TFileList は，以下のコマンドをブロードキャストします。

表 15.4
TFileList のコマンド

定数	値	意味
cmFileFocused	102	***TFileList*** オブジェクトで，新しいファイルにフォーカスがきた。
cmFileDoubleClicked	103	***TFileList*** オブジェクトで，ファイルが選択された。

メンバ関数

コンストラクタ　TFileList(const TRect& bounds, const char *aWildCard, TScrollBar *aScrollBar);

TSortedListBox コンストラクタを呼び出し，指定された境界と *aScrollBar* がゼロでない場合には，垂直スクロールバーを持つ 2 段の ***TFileList*** オブジェクトを作成します。引数 *aWildCard* は，ドライブ，パス，ファイル，および拡張子(DOS 標準のワイルドカードは，展開されます) を指定する文字列であり，ファイルコレクションはそれを基にして ***readDirectory*** を呼び出すことで生成されます。

デストラクタ　TFileList ( );

ファイルリストを消去します。

build　static TStreamable *build ( );

ストリーム読み出しを行なうときに，オブジェクトを作成するために呼び出されます。

関連項目　***TStreamableClass***, ***ipstream::readData***

focusItem　virtual void focusItem (short item);

リスト内の指定された要素にフォーカスを与えます。***TSortedListBox::focusItem*** を呼び出し，***cmFileFocused*** イベントをブロードキャストします。

関連項目　***TSortedListBox::focusItem***

getText　virtual void getText (char *dest, short item, short maxLen);

item で指定された位置にある ***TSearchRec*** オブジェクトを取り出し，*dest* にファイル名を設定します。ファイル名がディレクトリであれば，¥¥ を付加します。

handleEvent virtual void handleEvent (TEvent& event) ;

TSortedListBox のイベントハンドラをそのまま継承します。***handleEvent*** は，通常のリストボックスイベントをすべて処理します。すなわち，マウスまたはキーボードカーソルによる移動に対して，ファイル名の選択およびハイライトを行ないます。

関連項目 ***TSortedListBox::handleEvent***，***TListBox::handleEvent***

list TFileCollection *list ( ) ;

プライベートデータメンバ ***items***，すなわち，このファイルリストボックスに現在リンクしている ***TFileCollection*** オブジェクトへのポインタを返します。

関連項目 ***TSortedListBox::list***，***TListBox::items***

newList void newList (TFileCollection *aList) ;

TSortedListBox::newList を呼び出し，それまでこのリストボックスにリンクされていた ***TFileCollection*** オブジェクトを消去し，*aList* で指定されたファイルコレクションをそのかわりとします。新しいコレクションでは，最初の要素にフォーカスが与えられます。

関連項目 ***TSortedListBox::newList***

read virtual void *read (ipstream& is) ;

入力ストリーム *is* からの読み出しを行ないます。

関連項目 ***TStreamableClass***，***ipstream***

readDirectory void readDirectory (const char *dir, const char *wildCard) ;
void readDirectory (const char *wildCard) ;

wildCard 文字列を展開してファイルコレクションを生成し，このファイルリストにリンクします。最初の形式では，*dir* 引数で相対パスまたは絶対パスを別個に指定することができます。サブディレクトリの区切文字には，/ または ¥ を使うことができます（ただし，/ も ¥ にして出力されます）。生成された ***TFileCollection*** オブ

ジェクト(***TSearchRec*** オブジェクトのソート済みの集合)は，プライベートデータメンバ ***items***（メンバ関数 ***list*** を介してアクセスできる）に代入されます。生成されるファイルが多すぎる場合，警告を行なうメッセージボックスを表示します。***readDirectory*** は，ファイル属性についても考慮し，隠しファイルの名前を生成することはありません。

関連項目　***TFileList::TFileList***

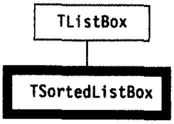
write　virtual void write (opstream& os) ;

出力ストリーム *os* への書き込みを行ないます。

関連項目　***TStreamableClass***, ***opstream***

関連する関数

演算子関数で ***TFileList*** に関係するものもありますが，メンバ関数とはなっていません。詳細については第 13 章の322 ページを参照してください。



TSortedListBox は ***TListBox*** を特別化した導出クラスであり，その要素をソートして管理します。***TFileList*** などのリストボックスクラスのベースとして定義されたものです。

メンバ関数

コンストラクタ　TSortedListBox(const TRect& bounds, ushort aNumCols, TScrollBar * aScrollBar) ;

TListBox::TListBox (*bounds, aNumCols, aScrollBar*) を呼び出し，指定されたサイズ，段数，および垂直スクロールバーを備えたリストボックスを作成します。***shiftState*** を 0 にし，カーソルは最初の要素のところに設定します。

getKey　virtual void *getKey (const char *s) ;　　private

すべての導出クラスで，このプライベートメンバ関数を，指定の文字列 *s* に対してキーを返すように定義しなくてはなりません。定義は，導出クラスで採用しているソートの方法に応じて異なります。デフォルト処理として，***getKey*** は *s* を返します。

handleEvent　virtual void handleEvent (TEvent& event) ;

TListBox::handleEvent を呼び出してから，リストから要素を選択するために使用される特殊なキーイベントおよびマウスイベントを処理します。

関連項目　***TListBox::handleEvent***

list TSortedCollection *list ( ) ;

このソート済みのリストボックスに現在リンクしている ***TSortedCollection*** オブジェクトへのポインタを返します。この関数により，プライベートのデータメンバ ***items***，すなわちリスト表示されたり選択されたりする要素へのポインタにアクセスすることができます。導出されるソート済みのリストボックスクラスでは，通常 *list* をオーバーライドし，***TSortedCollection*** の導出クラスのオブジェクトへのポインタを返すようにします。たとえば，***TFileList::list*** は，***TFileList::items*** で指定される ***TFileCollection*** オブジェクトへのポインタを返します。

関連項目 ***TSortedListBox::list***，***TListBox::list***，***TFileList::list***

newList void newList (TSortedCollection *aList) ;

TListBox::newList を呼び出し，それまでこのリストボックスにリンクされていた ***TSortedCollection*** オブジェクトを消去し，*aList* で指定されたコレクションをそのかわりとします。新しいコレクションでは，最初の要素にフォーカスが与えられます。

関連項目 ***TListBox::newList***

第16章
グローバルリファレンス

標準クラスについては，第13章『クラスリファレンス』で説明しています。また，エディタクラスとダイアログクラスについては，それぞれ第14章と第15章で説明しています。

この章では，Turbo Vision の標準的なクラス階層の中では定義されていない Turbo Vision の要素すべてについて解説します。この章で記載する要素は，Turbo Vision の各ヘッダファイルで定義されている typedef，列挙型，定数，変数，およびメンバとなっていない関数（グローバル関数）です。リファレンスの標準的な構成は次のとおりです。

sample 関数 SAMPLE.H

宣言 void sample (int arg) ;

機能 ***sample*** は，***int*** 型の引数 *arg* をとり，ある種の機能を実行します。

関連項目 関連するクラス，関数など。

apXXXX 定数 APP.H

値 アプリケーションパレット定数の定義は以下のとおりです。

表 16.1
アプリケーション
パレット定数

定数	値	意味
apNColor	0	ノーマルアトリビュート用のパレットを使用
apMono	1	LCD 画面用のパレットを使用
apEColor	2	拡張アトリビュート用のパレットを使用

機能 *ap* で始まる定数は, Turbo Vision アプリケーションで使用される 3 種類の標準カラーパレットを指定するために用いられます。3 種類のパレットは, それぞれノーマルアトリビュート用, 白黒ディスプレイ用, 拡張アトリビュート用として使用されます。

Boolean 列挙型 TTYPES.H

宣言 enum Boolean { False, True };

機能 *False* に *int* 値 0 を, *True* に 1 を割り当てます。if (tesfunc ( )) {...} というテストと if (True == testfunc ( )) {...} というテストが, 等しくない場合があるということに注意してください。

BUILDER typedef TOBJSTRM.H

宣言 typedef TStreamable * (*BUILDER) ( );

機能 ストリーム入出力可能なクラスは, それぞれ *BUILDER* 型のビルダ (組み立て関数) を用意しています。ビルダは, ストリーム読み出し処理でオブジェクトを作成するときに, 適当なサイズの生のメモリを割り当て, vtable ポインタを初期化します。ストリーム入出力可能なクラスの *read* 関数は, ストリームからデータを読み出し, ビルダが作成したオブジェクトに格納します。

関連項目 *TView::build*, *TStreamable*

bfXXXX 定数　DIALOGS.H

値　ボタンフラグの定義は以下のとおりです。

表 16.2 ボタンフラグ

定数	値	意味
bfNormal	0x00	ボタンをノーマルボタンにする
bfDefault	0x01	ボタンをデフォルトボタンにする
bfLeftJust	0x02	ボタンのラベルを左詰めにする

機能　新しく作り出すボタンのスタイルを決定する際に，これらの値を組み合わせたものが *aFlags* 引数で ***TButton*** コンストラクタに渡されます。***bfNormal*** は，通常のデフォルトではないボタンを指定します。***bfDefault*** は，ボタンをデフォルトのボタンにすることを指定します。プログラムでは，***TGroup*** にデフォルトボタンが1個しかないようにしなければなりません。***bfLeftJust*** の値は，***bfNormal*** あるいは ***bfDefault*** と組み合わせることができ，ボタン内に表示されるテキストの位置の指定を行ないます。クリアすればラベルは中央に置かれ，セットすれば左詰めにして表示されます。

関連項目　***TButton::flags***，***TButton::makeDefault***，***TButton::draw***

ccAppFunc typedef　TTYPES.H

宣言　typedef void (*ccAppFunc)(void *item, void *arg) ;

機能　繰り返し処理関数で，コレクション内の一定の範囲の要素に対して適用される処理関数および引数リストを指定するときに使用されます。

関連項目　***TNSCollection::forEach***，***ccTestFunc***

ccIndex typedef　TTYPES.H

宣言　typedef int ccIndex ;

機能　コレクション中の要素をインデックスしたり，カウントするときに使用されます。

関連項目　***TNSCollection::at***

ccNotFound 定数　TTYPES.H

宣言　const ccNotFound = −1 ;

機能　各種コレクション検索関数が，検索に失敗したときに返す ***ccIndex*** 型の値です。

関連項目　***TNSCollection::indexOf***

ccTestFunc typedef　TTYPES.H

宣言　typedef Boolean (*ccTestFunc)(void *item, void *arg) ;

機能　繰り返し処理関数で，コレクション内の一定の範囲の要素に対して適用される処理関数および引数リストを指定するときに使用されます。

関連項目　***TNSCollection::firstThat***, ***TNSCollection::lastThat***, ***ccAppFunc***

cmXXXX 定数 VIEWS.H

機能 これらの定数は，Turbo Vision の定義済みのコマンドを意味します。これらのコマンドは，***evMessage*** イベント（ ***evCommand*** と ***evBroadcast*** ）の ***TEvent::command*** データメンバに入れられ，Turbo Vision の標準オブジェクトの ***handleEvent*** メソッドに，いろいろな仕事を実行させる機能を持ちます。

Turbo Vision は，Turbo Vision 自体が使用するコマンドとして，0～99 と 256～999 定数値を予約しています。Turbo Vision 標準のオブジェクトのイベントハンドラが，これらの定義済み定数の処理を行ないます。100～255 および 1000～65,535 の範囲の定数を定義することによって，定義済みコマンドと衝突させずにアプリケーション独自の定数を定義することができます。

値 Turbo Vision では，**表 16.3** に示した標準コマンドが定義されており，Turbo Vision の標準オブジェクトで使用されます。

表 16.3 標準コマンドコード

コマンド	値	意味
cmValid	0	新たにインスタンス化したビューの有効性をチェックするために ***TView::valid*** に渡します。
cmQuit	1	***TProgram::handleEvent*** に ***endModal*** (***cmQuit***) を呼び出させ，アプリケーションを終了させます。通常は，ステータスラインかメニューのどこかに ***kbAltX*** を ***cmQuit*** に対応づけるエントリが置かれます。
cmError	2	どのオブジェクトからも処理されません。未実装のコマンドやサポートされていないコマンドを表わすためなどに使用します。
cmMenu	3	***TMenuView::handleEvent*** に，自分自身の ***execView*** を呼び出してメニュー選択処理を行なわせるためのコマンドです。その結果，***putEvent*** によって新しいコマンドが発行されたりすることになります。通常は，スタータスラインに ***kbF10*** を ***cmMenu*** に対応づけるエントリが置かれます。
cmClose	4	イベントオブジェクトの ***infoPtr*** データメンバ が 0 であるか，ウィンドウを指しているときに，***TWindow::handleEvent*** によって処理されます。ウィンドウがモーダル（たとえば，モードつきダイアログボックス）であれば，***cmCancel*** の値をもつ ***evCommand*** イベントが ***putEvent*** によって発行されます。ウィンドウがモードなしである場合には，ウィンドウがクローズをサポートしていれば (***wfClose*** フラグを参照)，ウィンドウの ***close*** メンバ関数が呼び出されます。ウィンドウのクローズボックスをクリックすれば，コマンドが ***cmClose*** で，***InfoPtr*** がウィンドウを指す ***evCommand*** イベントが発行されます。通常は，ステータスラインかメニューのどこかに，***kbAltF3*** を ***cmClose*** に対応づけるエントリが置かれます。
cmZoom	5	ウィンドウがズームをサポートしており(***wfZoom*** フラグを参照)，イベントオブジェクトの ***infoPtr*** データメンバが 0 であるかウィンドウを指している場合に，***TWindow::handleEvent*** に 自分自身の ***TWindow::zoom*** を呼び出させます。ウィンドウのズームボックスをクリックするか，ウィンドウのタイトルバーをダブルクリックすれば，コマンドが ***cmZoom*** で，***infoPtr*** がそのウィンドウを指す ***evCommand*** イベントが発行されます。通常は，ステータスラインかメニューのどこかに，***kbF5*** を ***cmZoom*** に対応づけるエントリが置かれます。
cmResize	6	ウィンドウがリサイズをサポートしている場合 (***wfMove***, ***wfGrow*** フラグを参照) に，***TWindow::handleEvent*** に自分自身の ***TView::dragView*** を呼び出させます。通常は，ステータスラインかメニューのどこかに，***kbCtrlF5*** を ***cmResize*** に対応づけるエントリが置かれます。

cmNext	7	デスクトップ上の一番最後のウィンドウを他のすべてのウィンドウよりも前面にくるように，***TDeskTop::handleEvent*** に移動を行なわせます。通常は，ステータスラインかメニューのどこかに，***kbF6*** を ***cmNext*** に対応づけるエントリが置かれます。
cmPrev	8	デスクトップ上の一番最初のウィンドウが，他のすべてのウィンドウの背後にいくように，***TDeskTop::HandleEvent*** に移動を行なわせます。通常は，ステータスラインかメニューのどこかに，***kbShiftF6*** を ***cmPrev*** に対応づけるエントリが置かれます。

ダイアログボックスオブジェクトのデフォルト動作は，**表 16.4** に示した標準コマンドで定義されています。

表 16.4　ダイアログボックスの標準コマンド

コマンド	値	意味
cmOK	10	OK ボタンが押された
cmCancel	11	Cancel ボタン，クローズアイコン，もしくは ***ESC*** キーで，ダイアログボックスがキャンセルされた
cmYes	12	Yes ボタンが押された
cmNo	13	No　ボタンが押された
cmDefault	14	デフォルトボタンが押された

cmOK，***cmCancel***，***cmYes***，あるいは ***cmNo*** のいずれかのコマンドを備えたイベントは，モードつきダイアログ の ***TDialog::handleEvent*** にダイアログを終了させ，そのコマンドの値を戻させます（***endModal*** を呼び出して）。通常，モードつきダイアログには，これらのコマンド値のどれかを備える ***TButton*** が少なくとも 1 個置かれます。***TDialog::handleEvent*** は，***kbEsc*** キーボードイベントを受け取ると，***cmCancel*** コマンドイベントを発行します。

cmDefault コマンドは，デフォルトボタン（***bfDefault*** フラグを参照）の ***TButton::handleEvent*** にボタン押下をシミュレートさせます。***TDialog::handleEvent*** は，***kbEnter*** キーボードイベントを受け取ると，***cmDefault*** コマンドイベントを発行します。

標準ビューおよびエディタアプリケーションで使用されるコマンドとして，**表 16.5**（次頁）に示す標準コマンドが定義されています。

表 16.5　標準ビューコマンド

コマンド	値	意味
cmCut	20	エディタコマンドコード
cmCopy	21	
cmPaste	22	
cmUndo	23	
cmClear	24	
cmTile	25	
cmCascade	26	
cmReceivedFocus	50	***sfFocused*** が変更されるたびに，***TView::setState*** が ***message*** 関数を使用して，この2つの値のどちらかをもつ ***evBroadcast*** イベントを自分自身の ***TView::owner*** に送ります。そのイベントの ***infoPtr*** は，そのビュー自身を指します。このイベントの実際上の機能は，そのビューがフォーカスを受け取ったり解放したりしたことを同列ビューに知らせ，それらのビューに自分自身を適宜更新させるというものです。たとえば，***TLabel*** オブジェクトによってラベルを付けられた同列ビューがフォーカスを獲得したりフォーカスを放棄した場合，***TLabel*** クラスがこのコマンドを受け取ることで自分自身をハイライトしたりハイライトをやめたりします。
cmReleasedFocus	51	
cmCommandSetChanged	52	***TProgram::idle*** メンバ関数が，現在のコマンドセットに変更があったこと(***TView*** の ***enableCommands***, ***disableCommands***, あるいは ***setCommands*** メソッドを呼び出すことによって変更が起こる)を検出したときに，この値の ***evBroadcast*** イベントを発行します。***cmCommandSetChanged*** ブロードキャストは，物理的な階層（実際にインスタンス化されたクラス階層）中のすべてのビュー（***eventMask*** でとくに ***evBroadcast*** イベントをマスクしているビューは除く)の ***handleEvent*** に送られます。コマンドセットを変更することによってビューの形態に変化が起こる場合には，そのビューは，***cmCommandSetChanged*** を受け取ったときに自分自身を再表示する必要があります。たとえば，***TButton***, ***TMenuView***, ***TStatusLine*** の各オブジェクトは，このコマンドを受け取った場合は自分自身の再表示を行ないます。
cmScrollBarChanged	53	***TScrollBar*** の値が変化するか，スクロールバー上でマウスのクリックがあった場合に，***TScrollBar*** が，***message*** 関数を使って，この2つの中のどちらかの値の ***evBroadcast*** イベントをそのオーナーに送ります。そのイベントの ***infoPtr*** は，スクロールバー自身を指します。***TScroller*** や ***TListViewer*** クラスなど，スクロールバーによって制御される同列ビューが，この種のブロードキャストの処理を行ないます。
cmScrollBarClicked	54	

cmSelectWindowNum	55	イベントオブジェクトの ***infoInt*** が ***TWindow::Number*** に一致する場合，***TWindow::handleEvent*** に自分自身の ***TView::select*** を呼び出させます。***TProgram::handleEvent*** は，**GRPH-1** から **GRPH-9** までのキーボードイベントを受け取ると，***infoInt*** が 1 から 9 までの ***cmSelectWindowNum*** イベントをブロードキャストします。
cmListItemSelected	56	要素が選択されたことを表わす ***TListViewer*** のメッセージ。
cmRecordHistory	60	***THistory*** クラスに，それが管理している ***TInputLine*** の現在の内容を「記録」させます。***TButton*** は，ボタンが押されたときに，オーナーに対して ***cmRecordHistory*** ブロードキャストを送りますが，これは実際には，ダイアログ内のすべての ***THistory*** オブジェクトにその時点で「記録を行な」わせることになります。

関連項目 ***TView::handleEvent***， ***TCommandSet***

cstrlen 関数　　UTIL.H

宣言 int cstrlen (const char *s) ;

機能 文字列 *s* の長さを戻します。ここで *s* は，ホットキーを指定するためのチルダ文字 '~' を含んだ制御文字列です。'~' は画面には表示されないので，文字列の長さには含めません。たとえば，引数に '~B~roccoli' を指定した場合には，***cstrlen*** は 8 を返します。

関連項目 ***moveCStr***

ctrlToArrow 関数　UTIL.H

宣言　ushort ctrlToArrow (ushort keyCode) ;

機能　WordStar と同じコントロールキーコードを，対応するカーソルキーコードに変換します。*keyCode* の下位バイトが，**表 16.6** にあるコントロールキーの値のどれかと一致すれば，結果(返される値)は，それに対応する ***kbXXXX*** 定数となります。一致しない場合には，*keyCode* がそのまま返されます。

表 16.6
コントロールキーのマッピング

キー入力	キーコードの下位バイト	結果
CTRL-A	0x01	***kbHome***
CTRL-D	0x04	***kbRight***
CTRL-E	0x05	***kbUp***
CTRL-F	0x06	***kbEnd***
CTRL-G	0x07	***kbDel***
CTRL-S	0x13	***kbLeft***
CTRL-V	0x16	***kbIns***
CTRL-X	0x18	***kbDown***

DEFAULT_SAFETY_POOL_SIZE　BUFFERS.H

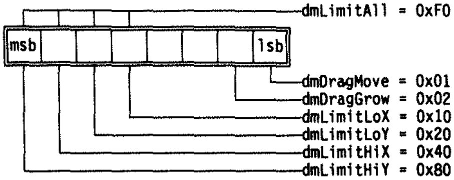
宣言　const DEFAULT_SAFETY_POOL_SIZE = 8192 ;

機能　セーフティプールのデフォルトの初期サイズをバイト単位で与えます。この宣言を書き換えて，この値を変更することもできます。引数でサイズを指定せずに ***TVMemMgr::resizeSafetyPool*** を呼び出した場合，このデフォルト値が使用されます。

関連項目　***TVMemMgr::resizeSafetyPool***

値　*dragMode* ビットの定義は以下のとおりです。

図 16.1
dragMode の
ビットフラグ



機能　ドラッグモード定数は，***TView::dragView*** の引数 *mode* を指定する際に使用されます。ビューの移動やサイズの変更を許すかどうか，および引数 *limits* をどのように解釈するかを，これらのフラグで指定します。

ドラッグモード定数の定義は以下のとおりです。

表 16.8
ドラッグモード定数

定数	意味
dmDragMove	ビューの移動を許す
dmDragGrow	ビューのサイズ変更を許す
dmLimitLoX	ビューの左辺を *limits* より外にもっていくことはできない
dmLimitLoY	ビューの上辺を *limits* より外にもっていくことはできない
dmLimitHiX	ビューの右辺を *limits* より外にもっていくことはできない
dmLimitHiY	ビューの下辺を *limits* より外にもっていくことはできない
dmLimitAll	ビューのどの部分も *limits* より外にもっていくことはできない

TView の ***dragMode*** データメンバには，***dmLimitXX*** フラグをどのように組み合わせたものでも指定することができます。デフォルトでは，***TView*** コンストラクタがデータメンバに ***dmLimitLoY*** をセットします。現在のところ ***dragMode*** データメンバは，***TWindow::dragView*** がウィンドウを移動したりリサイズしたりするときにしか使用されていません。

EOS 定数　　TTYPES.H

宣言　const char EOS = '¥0';

機能　文字列の最後のヌルと同義です。

eventQSize 定数　　CONFIG.H

宣言　const eventQSize = 16;

機能　イベントキューのサイズを指定します。

evXXXX 定数　　SYSTEM.H

機能　Turbo Vision のイベントハンドラにイベントの種類を知らせるためのニーモニックです。***evXXXX*** 定数は，イベント構造体の ***event*** データメンバ，ビューオブジェクトの ***eventMask*** データメンバ，および ***positionalEvents*** 変数，***focusedEvents*** 変数で使用されています。

値　標準的なイベントの種類は，以下のイベントフラグを用いて指定されます。

表 16.8
標準のイベントフラグ

定数	値	意味
evMouseDown	0x0001	マウスボタンが押された
evMouseUp	0x0002	マウスボタンが放された
evMouseMove	0x0004	マウスが位置を変えた
evMouseAuto	0x0008	マウスボタンが押されつづけているときに周期的に発生するイベント
evKeyDown	0x0010	キーが押された
evCommand	0x0100	コマンドイベント
evBroadcast	0x0200	ブロードキャストイベント

イベントの種類は，以下の定数を用いてマスクすることができます。

表 16.9
標準のイベントマスク

定数	値	意味
evNothing	0x0000	イベントは処理済みである
evMouse	0x000F	マウスイベント
evKeyboard	0x0010	キーボードイベント
evMessage	0xFF00	メッセージイベント（コマンド，ブロードキャスト，またはユーザ定義のイベント）

イベントマスクビットの定義は，以下のとおりです。

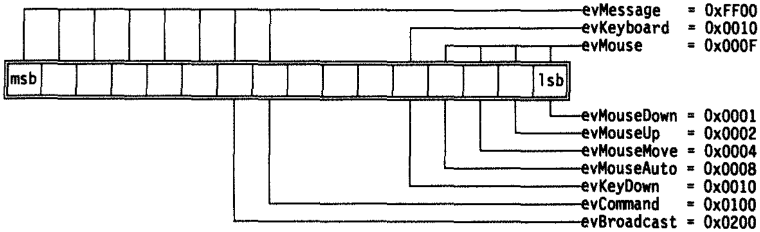


図 16.2
イベントマスクのビットマッピング

標準イベントマスクを用いて，あるイベントが特定のイベント「ファミリー」に属しているかどうかを判定することができます。たとえば以下のようにします。

```
if ( (event.what & evMouse != 0 ) doMouseEvent();
```

関連項目 ***TEvent***, ***TView::eventMask***，***getKeyEvent***，***getMouseEvent***, ***handleEvent***，***positionalEvents***，***focusedEvents***

focusedEvents 定数 VIEWS.H

宣言 focusedEvents = evKeyboard | evCommand ;

機能 フォーカスイベントとなっているイベントクラスを定義します。***focusedEvents*** と ***positionalEvents*** は，***TGroup::handleEvent*** がイベントをグループのサブビューにどういうルートで渡すかを指定するときに使用します。あるイベントクラスが，***focusedEvents*** にも ***positionalEvents*** にも含まれていない場合には，そのイベントはブロードキャストイベントとして扱われます。

関連項目 ***positionalEvents***, ***TGroup::handleEvent***, ***TEvent***, ***evXXXX***

genRefs 関数 GENINC.H

宣言 void genRefs ( ) ;

GENINC.CPP が，いろいろなクラスのデータメンバのアセンブラでのオフセットを生成するために，***genRefs*** を使用します。GENINC.EXE は，TV.LIB の構築に必要な TVWRITE.INC ファイルを作成します。***genRefs*** は，***TDrawBuffer***, ***TEditor***, ***TEventQueue***, ***TTerminal***, ***TView***, および ***TGroup*** のフレンド関数となっています。上級ユーザが，自前の Turbo Vision ライブラリを開発しようとする場合に必要とされるものです。

getAltChar 関数 UTIL.H

宣言 char getAltChar (ushort keyCode) ;

機能 ***GRPH****-ch* によって，引数 *keyCode* で指定した 2 バイトのスキャンコードを発生するような文字 *ch* を返します。これは，***getAltCode*** と逆の対応付けを行なう関数です。

関連項目 ***getAltCode***

getAltCode 関数　UTIL.H

宣言　ushort getAltCode (char ch) ;

機能　***GRPH**-ch* に対応する 2 バイトのスキャンコード（キーコード）を返します。これは，***getAltChar*** と逆の対応付けを行なう関数です。

関連項目　***getAltChar***

gfXXXX 定数　VIEWS.H

機能　***TView*** および ***TView*** のすべての導出オブジェクトの ***growMode*** データメンバを設定する際に使用されるニーモニックです。***growMode*** の各ビットを設定することによって，オーナーのサイズに変更があったときに，ビューのサイズをどのように変更するかが決定されます。

値　***growMode*** ビットの定義は，以下のとおりです。

図 16.3
growMode の
ビットマッピング

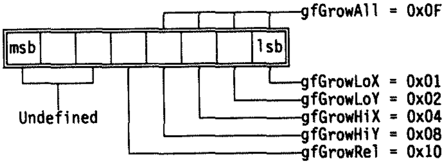


表 16.10
グローモードフラグの定義

LoX = 左辺
LoY = 上辺
HiX = 右辺
HiY = 下辺

定数	意味
gfGrowLoX	立っていれば，ビューの左辺はオーナーの右辺と一定の距離を保つ。立っていなければ，そのような移動は起こらない。
gfGrowLoY	立っていれば，ビューの上辺がオーナーの下辺と一定の距離を保つ。
gfGrowHiX	立っていれば，ビューの右辺がオーナーの右辺と一定の距離を保つ。
gfGrowHiY	立っていれば，ビューの下辺がオーナーの下辺と一定の距離を保つ。
gfGrowAll	立っていれば，ビューは，オーナーの右下隅に合わせて移動する。
gfGrowRel	デスクトップ内の TWindow オブジェクトで使用される。ビューは，オーナーのサイズにしたがってそのサイズを変更する。ウィンドウは，25 行モードと 31 行モードを切り替えたときでも，オーナーとの相対的なサイズを維持する。

関連項目　***TView::growMode***

hcXXXX 定数　　VIEWS.H

値　ヘルプコンテキスト定数の定義は，以下のとおりです。

表 16.11
ヘルプコンテキスト定数

定数	値	意味
hcNoContext	0	コンテキストは指定されていない
hcDragging	1	オブジェクトは，現在ドラッグされている

機能　***TView::helpCtx*** のデフォルト値は，***hcNoContext*** です。これは，そのビューに対するヘルプコンテキストがないことを意味します。ビューがドラッグ中であれば(***sfDragging*** 状態フラグで示される)，***TView::getHelpCtx*** は，常に ***hcDragging*** を返します。

Turbo Vision は，自分で使用するヘルプコンテキストとして，0 から 999 までの値を予約しています。アプリケーション独自の定数

は，1000 から 65,535 までの範囲で定義することができます。

関連項目 *TView::getHelpCtx*，*TStatusLine::update*

historyAdd 関数　UTIL.H

宣言 void historyAdd (uchar id, const char *str) ;

機能 *id* で指定されるヒストリリストに文字列 *str* を追加します。

historyCount 関数　UTIL.H

宣言 ushort historyCount (uchar id) ;

機能 ID 番号 *id* に対応するヒストリーリスト内の文字列の数を返します。

historyStr 関数　UTIL.H

宣言 const char *historyStr (uchar id, int index) ;

機能 ID 番号 *id* に対応するヒストリーリスト内の *index* 番目の文字列を返します。

inputBox 関数　MSGBOX.H

宣言 ushort inputBox (const char *title, const char *aLabel,
char *s, uchar limit) ;

機能 指定のタイトルとラベルを持つ入力ボックスを表示します。最大 *limit* 個までの文字を入力し，文字列 *s* に格納します。

関連項目 *inputBoxRect*

inputBoxRect 関数　MSGBOX.H

宣言　ushort inputBoxRect (const TRect &bounds, const char *title, const char *aLabel, char *s, uchar limit) ;

機能　指定の境界，タイトルおよびラベルを持つ入力ボックスを表示します。最大 *limit* 個までの文字を入力し，文字列 *s* に格納します。

関連項目　***inputBox***

kbXXXX 定数　TKEYS.H

値　キーボード状態の定義は以下のとおりです。この定数を使用し，絶対番地 0x40：0x17 の 1 バイトに格納されているキーボードのシフト状態を調べることができます。たとえば，以下のようにします。

表 16.12 キーボードの状態とシフトマスク

定数	値	意味
kbRightShift	0x0001	右 ***SHIFT*** キーが現在押されていれば，セットされる(注)
kbLeftShift	0x0001	左 ***SHIFT*** キーが現在押されていれば，セットされる(注)
kbCapsState	0x0002	***CAPS*** ロックの状態になっていればセットされる
kbAltShift	0x0008	***GRPH*** キーが現在押されていれば，セットされる
kbCtrlShift	0x0010	***CTRL*** キーが現在押されていれば，セットされる

(注) PC-9801では，左右どちらかのシフトを押しても ***kbRightShift*** と ***kbLeftShift*** の両方がセットされる。

キーボードのスキャンコードの定義は，以下のとおりです。この定数を使用し，***evKeyDown*** イベントレコードの ***TEvent::keyCode*** データメンバを調べることができます。

表 16.13
GRPH-文字キーコード

定数	値	定数	値
kbAltA	0x1E00	*kbAltN*	0x3100
kbAltB	0x3000	*kbAltO*	0x1800
kbAltC	0x2E00	*kbAltP*	0x1900
kbAltD	0x2000	*kbAltQ*	0x1000
kbAltE	0x1200	*kbAltR*	0x1300
kbAltF	0x2100	*kbAltS*	0x1F00
kbAltG	0x2200	*kbAltT*	0x1400
kbAltH	0x2300	*kbAltU*	0x1600
kbAltI	0x1700	*kbAltV*	0x2F00
kbAltJ	0x2400	*kbAltW*	0x1100
kbAltK	0x2500	*kbAltX*	0x2D00
kbAltL	0x2600	*kbAltY*	0x1500
kbAltM	0x3200	*kbAltZ*	0x2C00

表 16.14
特殊キーコード

定数	値	定数	値
kbAltEqual	0x8300	*kbEnd*	0x4F00
kbAltMinus	0x8200	*kbEnter*	0x1C0D
kbAltSpace	0x0200	*kbEsc*	0x011B
kbBack	0x0E08	*kbGrayMinus*	0x4A2D
kbCtrlBack	0x0E7F	*kbHome*	0x4700
kbCtrlDel	0x0600	*kbIns*	0x5200
kbCtrlEnd	0x7500	*kbLeft*	0x4B00
kbCtrlEnter	0x1C0A	*kbNoKey*	0x0000
kbCtrlHome	0x7700	*kbPgDn*	0x5100
kbCtrlIns	0x0400	*kbPgUp*	0x4900
kbCtrlLeft	0x7300	*kbrayPlus*	0x4E2B
kbCtrlPgDn	0x7600	*kbRight*	0x4D00
kbCtrlPgUp	0x8400	*kbShiftDel*	0x0700
kbCtrlPrtSc	0x7200	*kbShiftIns*	0x0500
kbCtrlRight	0x7400	*kbShiftTab*	0x0F00
kbDel	0x5300	*kbTab*	0x0F09
kbDown	0x5000	*kbUp*	0x4800

表 16.15
GRPH-数字の
キーコード

定数	値	定数	値
kbAlt1	0x7800	*kbAlt6*	0x7D00
kbAlt2	0x7900	*kbAlt7*	0x7E00
kbAlt3	0x7A00	*kbAlt8*	0x7F00
kbAlt4	0x7B00	*kbAlt9*	0x8000
kbAlt5	0x7C00	*kbAlt0*	0x8100

表 16.16
ファンクションの
キーコード

定数	値	定数	値
kbF1	0x3B00	*kbF6*	0x4000
kbF2	0x3C00	*kbF7*	0x4100
kbF3	0x3D00	*kbF8*	0x4200
kbF4	0x3E00	*kbF9*	0x4300
kbF5	0x3F00	*kbF10*	0x4400

表 16.17
シフトファンクションの
キーコード

定数	値	定数	値
kbShiftF1	0x5400	*kbShiftF6*	0x5900
kbShiftF2	0x5500	*kbShiftF7*	0x5A00
kbShiftF3	0x5600	*kbShiftF8*	0x5B00
kbShiftF4	0x5700	*kbShiftF9*	0x5C00
kbShiftF5	0x5800	*kbShiftF10*	0x5D00

表 16.18
Ctrl-ファンクションの
キーコード

定数	値	定数	値
kbCtrlF1	0x5E00	*kbCtrlF6*	0x6300
kbCtrlF2	0x5F00	*kbCtrlF7*	0x6400
kbCtrlF3	0x6000	*kbCtrlF8*	0x6500
kbCtrlF4	0x6100	*kbCtrlF9*	0x6600
kbCtrlF5	0x6200	*kbCtrlF10*	0x6700

表 16.19
GRPH-ファンクションのキーコード

定数	値	定数	値
kbAltF1	0x6800	***kbAltF6***	0x6D00
kbAltF2	0x6900	***kbAltF7***	0x6E00
kbAltF3	0x6A00	***kbAltF8***	0x6F00
kbAltF4	0x6B00	***kbAltF9***	0x7000
kbAltF5	0x6C00	***kbAltF10***	0x7100

関連項目　***evKeyDown***, ***getKeyEvent***

lowMemory 関数　UTIL.H

宣言　Boolean lowMemory ( ) ;

機能　***TVMemMgr::safetyPoolExhausted*** を呼び出し，セーフティプールの状態を調べます。

関連項目　***TVMemMgr::safetyPoolExhausted***

maxCollectionSize 変数　CONFIG.H

宣言　const maxCollectionSize = (int)((65536uL − 16)/sizeof(void *)) ;

機能　***maxCollectionSize*** は，1つのコレクションに登録できる要素数の最大値を決定します。基本的には，64 K のメモリセグメントに入れることのできるポインタの数となります。

maxFindStrLen 定数　CONFIG.H

宣言　const maxFindStrLen　= 80 ;

機能　***TEditor*** アプリケーションにおける検索文字列の最大長を与えます。

maxReplaceStrLen 定数 CONFIG.H

宣言 const maxReplaceStrLen = 80;

機能 ***TEditor*** アプリケーションにおける置換文字列の最大長を与えます。

maxViewWidth 定数 VIEWS.H

宣言 maxViewWidth = 132;

機能 ビューの最大幅を設定します。

関連項目 ***TView::size***

mbXXXX 定数 SYSTEM.H

機能 これらの定数を使用して，***evMouse*** イベントレコードの ***TEvent::buttons*** データメンバを調べることができます。たとえば，以下のような使い方をします。

```
if ( (event.what == evMouseDown) &&  (event.buttons == mbLeftButton) )
   doLeftButtonDownAction();
```

値 定数の定義は以下のとおりです。

表 16.20
マウスボタン定数

定数	値	意味
mbLeftButton	0x01	左ボタンが押されたときにセットされる
mbRightButton	0x02	右ボタンが押されたときにセットされる

関連項目 ***getMouseEvent***

message 関数　UTIL.H

宣言　void *message (TView *receiver, ushort what,
ushort command, void *infoPtr) ;

機能　***message*** は，引数の *event*, *command*, および *infoPtr* を用いてコマンドイベントを作成し，receiver－>handleEvent を呼び出して，このコマンドイベントを処理します(可能な場合)。*receiver* が 0 であるか，イベントの処理に失敗した場合には，***message*** は 0 を返します。イベントの処理に成功した場合 (すなわち，***handleEvent*** が ***evNothing*** の *event.what* を返してきた場合) には，***message*** 関数は *event.infoPtr* を返します。後者の場合，発送したイベントをどのビューが実際に処理したのかを *event.infoPtr* から知ることができます。引数 *event* には，通常，***evBroadcast*** がセットされます。たとえば，デフォルトの ***TScrollBar::ScrollDraw*** は，以下のメッセージをスクロールバーのオーナーに発送します。

```
message( owner, evBroadcast, cmScrollBarChanged, this );
```

このメッセージは，スクロールバーの ***value*** が変更されるたびに，関係するビューを再表示することができるようにします。

関連項目　***TView::handleEvent***, ***TEvent***, ***cmXXXX***, ***evXXXX***

messageBox 関数　MSGBOX.H

宣言　ushort messageBox (const char *msg, ushort aOptions) ;ushort messageBox (ushort aOptions, const char *msg, ...) ;

機能　最初の形式の場合，指定されたメッセージを表示します。2 番目の形式の場合，*msg* を書式指定文字列とみなして，それ以降のパラメータを処理します。*aOptions* には，後出の ***mfXXXX*** の項に示されるメッセージボックス定数の中のいずれかを指定します。

関連項目　***mfXXXX***

messageBoxRect 関数　　MSGBOX.H

宣言　ushort messageBoxRect (const TRect &r, const char *msg,
ushort aOptions) ;
ushort messageBoxRect (const TRect &r, ushort aOptions,
const char *msg, ...) ;

機能　最初の形式の場合，指定されたメッセージを指定した矩形に表示します。2番目の形式の場合，*msg* を書式指定文字列とみなして，それ以降のパラメータを処理します。*aOptions* には，後出の ***mfXXXX*** の項に示されるメッセージボックス定数の中のいずれかを指定します。

関連項目　***mfXXXX***

mfXXXX 定数　　MSGBOX.H

値　***messageBox*** で使用するメッセージボックス定数の定義は以下のとおりです。

表 16.21
messageBox 定数

定数	値	意味
mfWarning	0x0000	警告ボックス (Warning box) を表示する
mfError	0x0001	エラーボックス (Error box) を表示する
mfInformation	0x0002	情報ボックス (Information Box) を表示する
mfConfirmation	0x0003	確認ボックス (Confirmation Box) を表示する
mfYesButton	0x0100	ダイアログに Yes ボタンを持たせる
mfNoButton	0x0200	ダイアログに No ボタンを持たせる
mfOKButton	0x0400	ダイアログに OK ボタンを持たせる
mfCancelButton	0x0800	ダイアログに Cancel ボタンを持たせる

Yes, No, Cancel からなる標準的なダイアログボックスは次のように定義します。

```
mfYesNoCancel   = mfYesButton ¦ mfNoButton ¦fCancelButton;
```

OK, Cancel からなる標準的なダイアログボックスは，次のように定義します。

```
mfOKCancel     = mfOKButton ¦ mfCancelButton;
```

関連項目　***messageBox***

newStr 関数　UTIL.H

宣言　char *newStr (const char *s) ;

機能　動的に文字列を作成します。*s* が空であれば，***newStr*** は 0 ポインタを返します。そうでない場合は，***strlen*** (*s*)+1 バイトを割り当て，そこに *s* をコピーし(最後に '¥0' を付けて)，第 1 バイトへのポインタを返します。***newStr*** で作成した文字列は，***delete*** で廃棄することができます。

機能　これらのニーモニックを使用して，*TView.options* データメンバのビット位置を参照することができます。あるビット位置を 1 にすれば，ビューがその属性を持つことになり，そのビット位置をクリアすれば，その属性はオフ，すなわち無効となります。たとえば，以下のような指定をします。

```
myWindow.options = ofTileable | ofSelectable;
```

値　オプションフラグの定義は，以下のとおりです。

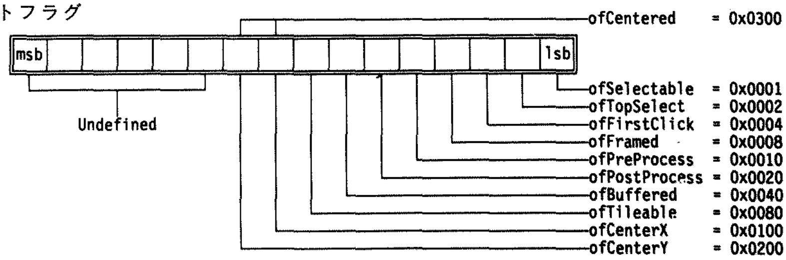
表 16.22　オプションフラグ

定数	意味
ofSelectable	たとえば，ビュー内でマウスクリックがあったり，ダイアログボックス内で **TAB** 入力があったときに，ビューに自動的に自分自身を選択させたい場合(*sfSelected* を参照) にセットします。
ofTopSelect	ビューが選択されたときに，他のすべての同列ビューよりも前面にそのビューを移動させたい場合にセットします。*ofTopSelect* ビットがセットされているときは，*TView::select* 呼び出しは *TView::makeFirst* に相当します。デフォルトでは，ウィンドウ (*TWindow* およびその下位オブジェクト)の ofTopSelect ビットはセットされます。したがって，そのウィンドウが選択された場合には，ディスクトップ上の他のすべてのウィンドウよりも手前に移されることになります。 〈関連項目〉 *TView::select*，*TGroup::makeFirst*
ofFirstClick	クリアされていれば，ビューを選択するためのマウスクリックは，ビューを選択すること以外には意味を持たないようになります。セットされていれば，そのようなマウスクリックは，ビューを選択した後も通常のマウスクリックとして処理されることになります。*ofSelectalbe* がセットされていない場合には，何の意味も持ちません。 〈関連項目〉 *TView::HandleEvent*，*sfSelect*，*ofSelectable*
ofFramed	ビューの周囲にフレームを表示したい場合にセットします。*TWindow* および *TWindow* の導出クラスは，一番最後のサブビューとして *TFrame* を所有します。*TFrame* が自分自体を表示するときには，*ofFramed* ビットのセットされている他のすべてのサブビューの周囲についてもフレームを表示します。 〈関連項目〉 *TFrame*，*TWindow*

ofPreProcess	フォーカスイベントがフォーカスビューに送られる前に，そのフォーカスイベントを受け取りたい場合にセットします。そうでなければクリアします。 〈関連項目〉 ***sfFocused***，***ofPostProcess***，***TGroup::phase***
ofPostProcess	フォーカスビューがフォーカスイベントを処理できなかったときに，フォーカスイベントを受け取りたい場合にセットします。そうしない場合にはクリアします。 〈関連項目〉 ***sfFocused***，***ofPreProcess***，***TGroup::phase***
ofBufferd	***TGroup*** オブジェクトおよび ***TGroup*** の導出クラスだけが使用します。充分なメモリがあるときに，キャッシュバッファを割り当てたい場合にセットされます。グループバッファには，グループ全体の画面イメージが格納されます。これによって，グループの再表示を高速化することができます。バッファが設けられていない場合には，***TGroup::draw*** は，各サブビューの ***drawView*** メソッドを呼び出していきます。その後のメモリ割り当て呼び出しで充分なメモリが得られなくなったら，グループバッファは解放され，そのメモリが利用できるようになります。
ofTileable	デスクトップが，このビューをタイル表示（あるいはカスケード表示）してもよいときにセットされます。普通は，***TWindow*** オブジェクトだけが使用します。
ofCenterX	***TGroup::insert*** を使用してこのビューをグループに挿入する際に，ビューを *x* 軸方向に関してオーナーの中央に置きたいときにセットします。
ofCenterY	***TGroup::insert*** を使用してこのビューをグループに挿入する際に，ビューを *y* 軸方向に関してオーナーの中央に置きたいときにセットします。
ofCetered	***TGroup::insert*** を使用してこのビューをグループに挿入する際に，ビューを *x* 軸，*y* 軸の両方向に関してオーナーの中央に置きたいときにセットします。

options ビットの定義は，以下のとおりです。

図 16.4
options ビットフラグ



関連項目　***TView::options***

operator ＋（＋演算子） MENUS.H

宣言　TSubMenu& operator ＋ (TSubMenu& s, TMenuItem& i);
TSubMenu& operator ＋ (TSubMenu&s1,TSubMenu&s2)；
TStatusDef& operator ＋ (TStatusDef&s1,TStatusItem&s2)；
TStatusDef& operator ＋ (TStatusDef&s1,TStatusDef&s2)；

機能　これらの＋オーバーロード演算子は，***TSubMenu***，***TMenuItem***，***TStatusDef***，および ***TStatusItem*** の各コンストラクタが，ステータスラインやネストしたメニューを作成するときに使用します。

関連項目　第 2 章の 44 ページ，48 ページ

operator delete(delete 演算子） NEW.CPP

宣言　void *operator delete (void *blk) ;

機能　ヒープからの割り当てブロックを解放します。セーフティプールを使い切った場合には，***TVMemMgr::resizeSafetyPool*** を呼び出します。***TVMemMgr::resizeSafetyPool*** は，それまでのセーフティプールを解放し，新しいデフォルトのセーフティプール（通常 8,192 バイト）を割り当てます。

関連項目　***TVMemMgr::resizeSafetyPool***

operator new(new 演算子） NEW.CPP

宣言　void *operator new (sizet sz) ;

機能　ヒープから *sz* バイトのメモリを割り当てようとします。***new*** に成功すれば，割り当てメモリへのポインタを返します。メモリ割り当てに失敗した場合には，キャッシュバッファ一つずつ解放していき

(キャッシュバッファがあった場合)，メモリ割り当てのリトライを行ないます。これが失敗に終わり，かつセーフティプールも「使い尽くした」場合には，***new*** 呼び出しは異常終了します。セーフティプールを使い尽くしていない場合には，セーフティプールからメモリ割り当てを行なおうとします。この場合，そのメモリ割り当てが成功したか失敗したかに関わらず，***TVMemMgr::safetyPool*** ポインタを 0 にします(セーフティプールが使い尽くされたことを意味する)。***new*** がセーフティプールからのメモリ割り当てに成功した場合には，その割り当てメモリへのポインタを返します。失敗した場合には，***abort*** を呼び出します。***new*** 演算子は，***TBufListEntry*** のフレンド関数です。

関連項目 ***TVMemMgr::safetyPool***，***TVMemMgr::safetyPoolExhausted***，***TVMemMgr::resizeSafetyPool***

positionalEvents 定数　　VIEWS.H

宣言 positionalEvents = evMouse ;

機能 ポジショナルイベントであるイベントクラスを定義します。***focusedEvents*** マスクおよび ***positionalEvents*** マスクは，***TGroup::handleEvent*** がイベントをグループのサブビューにどういうルートで渡すかを指定するときに使用します。あるイベントクラスが，***focusedEvents*** にも ***positionalEvents*** にも含まれていない場合には，そのイベントはブロードキャストイベントとして扱われます。

関連項目 ***TGroup::handleEvent***，***TEvent***，***evXXXX***，***focusedEvents***

sbXXXX 定数

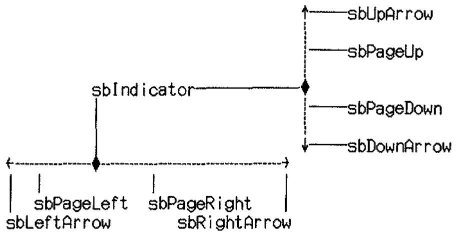
機能 これらの定数は，マウスクリックを行なうことのできる *TScrollBar* の各領域を定義します。

TScrollBar::scrollStep 関数が，これらの定数を実際のスクロールステップ値に変換します。*sbIndicator* 定数は，定義はされていますが，*TScrollBar::scrollStep* に渡されることはありません。

表 16.23 スクロールバーの部品定数

定数	値	意味
sbLeftArrow	0	水平スクロールバーの左向き矢印
sbRightArrow	1	水平スクロールバーの右向き矢印
sbPageLeft	2	水平スクロールバーの左ページ領域
sbPageRight	3	水平スクロールバーの右ページ領域
sbUpArrow	4	垂直スクロールバーの上向き矢印
sbDownArrow	5	垂直スクロールバーの下向き矢印
sbPageUp	6	垂直スクロールバーの上側ページ領域
sbPageDown	7	垂直スクロールバーの下側ページ領域
sbIndicator	8	スクロールバー上の位置インジケータ

図 16.5 スクロールバーの部品



TWindow::standardScrollBar 関数には，以下の値を渡すことができます。

表 16.24 standardScrollBar 定数

定数	値	意味
sbHorizontal	0x0000	スクロールバーは水平である
sbVertial	0x0001	スクロールバーは垂直である
sbHandleKeyboard	0x0002	スクロールバーはキーボードコマンドに対応する

関連項目 *TScrollBar*, *TScrollBar::scrollStep*, *TWindow::standardScrollBar*

selectMode 列挙型　　VIEWS.H

宣言 selectMode = {normalSelect, enterSelect, leaveSelect};

機能 Turbo Vision が内部的に使用します。

関連項目 *TGroup::execView*, *TGroup::setCurrent*

sfXXXX 定数　　VIEWS.H

機能 これらの定数を使用して，*TView::state* データメンバの各ビットにアクセスすることができます。*TView::state* は，決して直接変更しないでください。変更したい場合には，*TView::setState* を使用してください。

値 状態フラグの定義は，以下のとおりです。

S Globals

表 16.25　状態フラグ定数

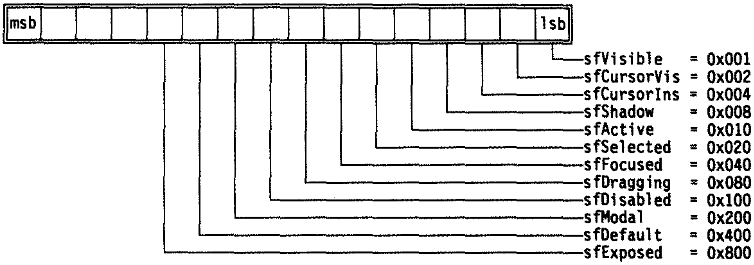
定数	意味
sfVisible	ビューがオーナー上に表示される場合にセットされます。そうでなければクリアされます。ビューは，デフォルトで ***sfVisible*** です。 ***TView::show*** および ***TView::hide*** を使用して，***sfVisible*** を変更することができます。ビューのオーナーが画面上に表示されているとはかぎりませんので，***sfVisible*** のビューが，必ずしも画面上に表示されるとはかぎりません。画面上に表示されるかどうかは，***sfExposed*** ビットを調べるか ***TView::exposed*** 関数を呼び出すかして判断してください。
sfCursorVis	ビューのカーソルが表示される場合にセットされます。そうでなければクリアされます。デフォルトはクリアです。***TView::showCursor*** および ***TView::hideCursor*** を使用して ***sfCursorVis*** を変更することができます。
sfCursorIns	ビューのカーソルがブロック型であればセットされます。ビューのカーソルがアンダーライン型（これがデフォルト）であればクリアされます。***TView::blockCursor*** および ***TView::normalCursor*** を使用し，***sfCursorIns*** を変更することができます。
sfShadow	ビューが影付きであればセットされます。そうでなければクリアされます。
sfActive	ビューがアクティブウィンドウであるか，またはアクティブウィンドウ内サブビューであればセットされます。
sfSelected	ビューが，そのオーナーの下で現在選択されているサブビューであればセットされます。***TGroup*** の各オブジェクトは，***current*** データメンバで，現在選択されているサブビューを指しています（どのサブビューも選択されていない場合は0)。1つの ***TGroup*** 内では，サブビューは，常にどれか1個だけしか選択されません。
sfFocused	ビューにフォーカスがきているときセットされます。ビューが選択されていて，そのビューの上位のオーナーがすべて選択されている場合，すなわち，***application*** (ビュー階層内の最上位のビュー）から始まるすべての ***TGroups*** の各 ***current*** ポインタによって形成されているチェーン上にそのビューがある場合に，そのビューにフォーカスがあります。フォーカスチェーン上の最後のビューが，すべてのフォーカスイベントの最終的な目的地となります。
sfDragging	ビューがドラッグされている最中であればセットされます。そうでなければクリアされます。
sfDisabled	ビューが無効ならセットされます。ビューが有効ならクリアされます。無効になっているビューは，そのビューに送られてくるすべてのイベントを無視することになります。

表 16.25　状態フラグ定数（続き）

sfModal	ビューがモードつきならセットされます。Turbo Vision のアプリケーションの実行中は，モードつきビューは，常に 1 個しか存在せず，通常は，***TApplication*** か ***TDailog*** オブジェクトがモードつきビューとなります。ビューが(***execView*** を呼び出して）実行を開始した場合，そのビューがモードつきビューとなります。モードつきビューは，有効なイベントのツリーの頂点（ルート）に位置し，その ***endModal*** メソッドが呼び出されるまで，イベントの取得と処理を行ないます。この「ローカルな」イベントループの最中には，イベントは，ビューツリーの下位のサブビューに渡されていきます。これらの下位のビューからのイベントは，ツリーに沿って遡って渡されていきますが，そのモードつきビューより上には渡されません。 〈関連項目〉 ***sfSelected***，***sfFocused***，***TView::setState***，***TView::handleEvent***，***TGroup::execView***
sfDefault	予備用のフラグです。ユーザ定義の状態を指定するために利用することができます。
sfExposed	ビューが，直接あるいは間接的に ***application*** オブジェクトに所有されており，したがって，画面に表示することができるものである場合にセットされます。***TView::exposed*** メンバ関数は，このフラグを参照し，さらにクリッピングの計算を行なうことによって，そのビューのどこかが実際に画面に表示されているかどうかを決定します。 〈関連項目〉 ***TView::exposed***

値　***state*** フラグビットの定義は以下のとおりです。

図 16.6
state フラグの
ビットマッピング



関連項目　***TView::state***

specialChars 変数　TTYPES.H

宣言　extern const ushort specialChars [] =

```
{
    '>' ,  ' , <' ,   0x1C,   0x1D, ' ', ' '
};
```

機能　モノクロビデオモードでフォーカスビューをハイライトにするために使用される強調指定文字を定義します。***ShowMarkers*** 変数が True であれば，これらの文字が表示されます。

関連項目　***showMarkers*** 変数

StreamableInit 列挙型　TTYPES.H

宣言　enum StreamableInit { streamableInit };

機能　ストリーム入出力可能なクラスには，それぞれ ***streamableInit*** を引数にとる特別のビルダコンストラクタが用意されています。ビルダコンストラクタ用の一義的なデータ型とするために，***streamableInit*** をこの列挙型で定義しています。

関連項目　TView::TView (streamableInit);

TScrollChars typedef　VIEWS.H

宣言　typedef char TScrollChars [5] ;

機能　***TScrollBar*** を表示するために使用される文字を表わす配列です。

関連項目　***TScrollBar***

uchar typedef　　TTYPES.H

宣言　typedef unsigned char uchar ;

機能　unsigned char と同義です。

ushort typedef　　TTYPES.H

宣言　typedef unsigned short ushort ;

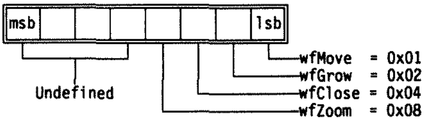
機能　unsigned short と同義です。

wfXXXX 定数　　VIEWS.H

機能　これらのニーモニックは，***TWindow*** オブジェクトの ***flags*** データメンバの各ビットを定義します。ビットがセットされれば，ウィンドウはそれに対応する属性をを持つことになります。ウィンドウには，移動可能，拡張可能，クローズ可能，ズーム可能という属性があります。

値　ウィンドウフラグの定義は以下のとおりです。

図 16.7
ウィンドウフラグ



U
Globals

表 16.26
ウィンドウフラグ定数

定数	値	意味
wfMove	0x01	ウィンドウは，移動することができる。
wfGrow	0x02	ウィンドウは，リサイズすることができ，右下隅にグローアイコンを持つ。
wfClose	0x04	ウィンドウフレームにクローズアイコンを持ち，そこをマウスクリックすることでウィンドウをクローズすることができる。
wfZoom	0x08	ウィンドウフレームにズームアイコンを持ち，そこをマウスクリックすることでウィンドウをズームすることができる。

あるビットをセット（=1）すれば，それに対応する特性が有効になり，クリア（=0）すれば，その特性は無効になります。

関連項目　***TWindows::flags***

wnNoNumber 定数　VIEWS.H

宣言　const ushort wnNoNumber = 0;

機能　***TWindow::number*** データメンバにこの定数を設定すると，そのウィンドウには番号付けがなされず，***GRPH*＋番号**キーでそのウィンドウを選択することができなくなります。***number*** データメンバに 1 から 9 までの数字が設定されれば，ウィンドウ番号が表示され，***GRPH*＋番号**で選択することができます。

関連項目　***TWindow::number***

write_args 構造体　VIEWS.H

宣言

```
struct writeargs {
void far *self ;
void far *target ;
void far *buf ;
ushort offset;
} ;
```

機能　***TView::writeView*** が内部的に使用します。

wpXXXX 定数　VIEWS.H

機能　これらの定数は，ウィンドウで使われる 3 種類の標準カラーマッピング割当てを定義します。デフォルトの ***TWindow::palette*** は ***wpBlueWindow*** です。***TDialog*** オブジェクトのデフォルトは，***wpGrayWindow*** です。

値　3 種類の標準ウィンドウパレットの定義は，以下のとおりです。

表 16.7
標準ウィンドウパレット

定数	値	意味
wpBlueWindow	0	ウィンドウテキストは，青地に黄色
wpCyanWindow	1	ウィンドウテキストは，シアン地に黄色
wpGrayWindow	2	ウィンドウテキストは，グレー地に黒

関連項目　***TWindow::palette***，***TWindow::getPalette***

索　引

【記号】

! 演算子
　pstream のメンバ関数　329
!= 演算子
　TPoint のフレンド　468
　TRect のメンバ関数　489
& 演算子
　TCommandSet のフレンド関数　371
&= 演算子
　TCommandSet のメンバ関数　371
+ 演算子
　TMenuItem と　674
　TPoint のフレンドと　468
　TStatusDef と　674
　TStatusItem と　674
　TSubMenu と　674
　オーバーロード　674
+= 演算子
　TCommandSet のメンバ関数　370
　TPoint のメンバ関数　468
- 演算子
　TPoint のフレンド　468
-= 演算子
　TCommandSet のメンバ関数　370
　TPoint のメンバ関数　468
<< 演算子
　opstream のフレンド　326
　ストリーム処理用にオーバーロードした　241
　ストリーム入出力可能なクラスと　323
　ヘッダファイル　242
= 演算子
　TPalette のメンバ関数　464
== 演算子
　TCommandSet のフレンド関数　371
　TPoint のフレンド　468
　TRect のメンバ関数　489
>> 演算子
　ipstream フレンド　319
　ストリーム処理用にオーバーロードした　241
　ストリーム入出力可能なクラスと　323
[] 演算子
　TPalette のメンバ関数　464
| 演算子
　TCommandSet friend　371
|= 演算子
　TCommandSet のメンバ関数　371

【A】

a
　TRect のデータメンバ　110, 488
amDefault
　TButton のデータメンバ　338
APP.H　291
appPalette
　TProgram のデータメンバ　472
apXXXX 定数　647
arStep
　TScrollBar のデータメンバ　502
at
　TDirCollection のメンバ関数　622
　TFileCollection のメンバ関数　628
　TGroup のメンバ関数　395
　TNSCollection のメンバ関数　219, 454
atInsert
　TDirCollection のメンバ関数　622
　TFileCollection のメンバ関数　629
　TNSCollection のメンバ関数　219, 454
atPut
　TDirCollection のメンバ関数　622
　TFileCollection のメンバ関数　629
　TNSCollection のメンバ関数　454
atRemove
　TNSCollection のメンバ関数　219, 454
attach
　fpbase のメンバ関数　313
autoIndent
　TEditor のデータメンバ　587

【B】

b

TRect のデータメンバ　110, 488
bad
pstream のメンバ関数　328
bakLabel
TColorDialog のデータメンバ　355
bakSel
TColorDialog のデータメンバ　355
basePos
TResourceFile のデータメンバ　494
bfBroadcast
ヘッダファイル　292
bfDefault
ヘッダファイル　292
bfLeftJust
ヘッダファイル　292
bfNormal
ヘッダファイル　292
bfXXXX 定数　649
ヘッダファイル　292
biosSeg
ヘッダファイル　296
blockCursor
TView のメンバ関数　551
Boolean
定義
ヘッダファイル　300
列挙型　648
bounds
入力ボックス　664
bp
pstream のデータメンバ　327
bufChar
TEditor のメンバ関数　592
bufDec
TTerminal のメンバ関数　541
buffer
TEditor のデータメンバ　587
TGroup のデータメンバ　392
TTerminal のデータメンバ　540
BUFFERS.H　291
bufInc
TTerminal のメンバ関数　541
bufLen
TEditor のデータメンバ　587
bufPtr
TEditor のメンバ関数　592
bufSize
TEditor のデータメンバ　587
TTerminal のデータメンバ　540
build
TAttrSelector のメンバ関数　332
TBackground のメンバ関数　335
TButton のメンバ関数　340
TChDirDialog のメンバ関数　619
TCheckBoxes のメンバ関数　344
TCluster のメンバ関数　348
TColorDialog のメンバ関数　357
TColorDisplay のメンバ関数　359
TColorGroupList のメンバ関数　363
TColorItemList のメンバ関数　366
TColorSelector のメンバ関数　368
TDeskTop のメンバ関数　374
TDialog のメンバ関数　378
TDirCollection のメンバ関数　622
TDirListBox のメンバ関数　626
TEditor のメンバ関数　592
TEditWindow のメンバ関数　607
TFileCollection のメンバ関数　629
TFileDialog のメンバ関数　635
TFileEditor のメンバ関数　610
TFileInfoPane のメンバ関数　638
TFileInputLine のメンバ関数　639
TFileList のメンバ関数　642
TFrame のメンバ関数　388
TGroup のメンバ関数　395
THistory のメンバ関数　408
TIndicator のメンバ関数　613
TInputLine のメンバ関数　420
TLabel のメンバ関数　425
TListBox のメンバ関数　429
TListViewer のメンバ関数　434
TMemo のメンバ関数　615
TMenuBar のメンバ関数　439
TMenuBox のメンバ関数　442
TMenuView のメンバ関数　445
TParamText のメンバ関数　466

TRadioButtons のメンバ関数　486
TResourceCollection のメンバ関数　491
TScrollBar のメンバ関数　504
TScroller のメンバ関数　510
TStaticText のメンバ関数　517
TStatusLine のメンバ関数　525
TStringCollection のメンバ関数　533
TStringList のメンバ関数　536
TStrListMaker のメンバ関数　538
TView のメンバ関数　551
TWindow のメンバ関数　574
コンストラクタ　247
ストリーム入出力可能なクラスへのリンク　251
独自の～の作成　255
BUILDER
typedef　648
ストリーム入出力可能なクラスと　530
ヘッダファイル　300
buttonCount
TMouse のデータメンバ　415

【C】

calcBounds
TView のメンバ関数　551
Cancel ボタン　26
コマンド　653
canInsert
TTerminal のメンバ関数　542
canUndo
TEditor のデータメンバ　587
cascade
TDeskTop のメンバ関数　374
ccAppFunc typedef　220, 649
ヘッダファイル　300
ccIndex typedef　218, 650
ヘッダファイル　300
ccNotFound ヘッダファイル　300
定数　650
ccTestFunc typedef　222, 650
ヘッダファイル　300
changeBounds
TEditor のメンバ関数　593
TGroup のメンバ関数　395
TListViewer のメンバ関数　434
TScroller のメンバ関数　510
TView のメンバ関数　551
charPos
TEditor のメンバ関数　593
charPtr
TEditor のメンバ関数　593
chars
TScrollBar のデータメンバ　502
CharScanType　311
ヘッダファイル　296
checkScrollBar
TEditor のメンバ関数　593
clear
pstream のメンバ関数　328
clearEvent
TView のメンバ関数　173, 191, 552
TView のメンバ関数
メッセージと　199
clearScreen
TDisplay のメンバ関数　381
TScreen のメンバ関数　501
clip
TGroup のデータメンバ　392
clipboard
TEditor のデータメンバ　587
clipCopy
TEditor のメンバ関数　593
clipCut
TEditor のメンバ関数　594
clipPaste
TEditor のメンバ関数　594
close
fpbase のメンバ関数　313
TEditWindow のメンバ関数　607
TWindow のメンバ関数　575
cmCancel
ヘッダファイル　304
cmCascade
ヘッダファイル　305
cmClear

ヘッダファイル 305
cmClose
ヘッダファイル 303
cmCommandSetChanged
ヘッダファイル 306
cmCopy
ヘッダファイル 305
cmCut
ヘッダファイル 305
cmDefault
ヘッダファイル 304
cmError
ヘッダファイル 303
cmHelp
ヘッダファイル 303
cmListItemSelected
ヘッダファイル 306
cmMenu
ヘッダファイル 303
cmNext
ヘッダファイル 303
cmNo
ヘッダファイル 304
cmOK
ヘッダファイル 304
cmPaste
ヘッダファイル 305
cmPrev
ヘッダファイル 303
cmQuit
ヘッダファイル 303
cmReceivedFocus
ヘッダファイル 306
cmRecordHistory
ヘッダファイル 292
cmReleasedFocus
ヘッダファイル 306
cmResize
ヘッダファイル 303
cmScrollBarChanged
ヘッダファイル 306
cmScrollBarClicked
ヘッダファイル 306
cmSelectWindowNum
ヘッダファイル 306
cmTile
ヘッダファイル 305
cmUndo
ヘッダファイル 305
cmValid
ヘッダファイル 303
cmXXXX 定数 76, 78, 184, 651
cmYes
ヘッダファイル 304
cmZoom
ヘッダファイル 303
colrSeg
ヘッダファイル 296
commandEnabled
TView のメンバ関数 552
commandSetChanged
TView のデータメンバ 545
compare
TFileCollection のメンバ関数 629
TNSSortedCollection のメンバ関数 459, 460
TSortedCollection のメンバ関数 514, 515
TStringCollection のメンバ関数 533
CONFIG.H 292
contains
TRect のメンバ関数 488
containsMouse
TView のメンバ関数 552
convertEvent
TEditor のメンバ関数 594
count
TNSCollection のデータメンバ 452
TResourceFile のメンバ関数 496
cpBackground パレット 336
cpBlueWindow パレット 578
cpButton パレット 342
cpCluster パレット 346, 351, 487
cpCyanWindow パレット 578
cpDialog パレット 379
cpFrame パレット 390

cpGrayWindow パレット 578
cpHistory パレット 410
cpHistoryViewer パレット 412
cpHistoryWindow パレット 414
cpInputLine パレット 423
cpLabel パレット 426
cpListViewer パレット 431, 438
cpMenuView パレット 440, 443, 448
cpScrollBar パレット 507
cpScroller パレット 512, 543, 544
cpStaticText パレット 467, 518
cpStatusLine パレット 527
createBackground
　TDeskInit のメンバ関数 372
createDeskTop
　TProgInit のメンバ関数 41, 471
createFrame
　TWindowInit のメンバ関数 579
createListViewer
　THistInit のメンバ関数 406
createMenuBar
　TProgInit のデータメンバ 41
　TProgInit のメンバ関数 471
createStatusLine
　TProgInit のデータメンバ 41
　TProgInit のメンバ関数 471
cstrlen グローバル関数 655
ctrlBreakHit
　TSystemError のデータメンバ 539
ctrlToArrow グローバル関数 656
curCommandSet
　TView のデータメンバ 546
curPos
　TEditor のデータメンバ 587
　TInputLine のデータメンバ 418
curPtr
　TEditor のデータメンバ 587
current
　TGroup のデータメンバ 393
　TMenuView のデータメンバ 444
cursorLines
　TScreen のデータメンバ 499
cursorVisible
　TEditor のメンバ関数 594
customization 260
　文字列リストと 267

【D】

data
　TDrawBuffer のメンバ関数 382
　TInputLine のデータメンバ 418
dataSize
　TChDirDialog のメンバ関数 619
　TCluster のメンバ関数 348
　TColorDialog のメンバ関数 357
　TGroup のメンバ関数 395
　TInputLine のメンバ関数 420
　TListBox のメンバ関数 429
　TMemo のメンバ関数 616
　TParamText のメンバ関数 466
　TView のメンバ関数 86, 552
DEFAULT_SAFETY_POOL_SIZE
　656
　ヘッダファイル 291
defs
　TStatusLine のデータメンバ 523
delCount
　TEditor のデータメンバ 588
delete
　destroy 対 79, 207, 462
delete 演算子 674
deleteRange
　TEditor のメンバ関数 594
deleteSelect
　TEditor のメンバ関数 595
delta
　TCollection のデータメンバ 216
　TEditor のデータメンバ 588
　TNSCollection のデータメンバ 218, 452
　TScroller のデータメンバ 508
　　～のオーバーライド 70
__DELTA マクロ
　ヘッダファイル 300
deskTop
　TProgram のデータメンバ 472

ポインタ　42
destroy
delete 対　79, 207, 462
new 演算子と　79
shutDown と　207, 463
TObject のメンバ関数　462
DIALOGS.H　292
dir
TDirEntry のメンバ関数　625
directory
TFileDialog のデータメンバ　633
disableCmd
TCommandSet のメンバ関数　370
disableCommand
TView のメンバ関数　553
disableCommands
TView のメンバ関数　553
display
TColorDialog のデータメンバ　355
dmDragGrow　155, 657
ヘッダファイル　304
dmDragMove　155, 657
ヘッダファイル　304
dmLimitAll　156
dmLimitHiX　156, 657
ヘッダファイル　304
dmLimitHiY　156, 657
ヘッダファイル　304
dmLimitLowX　155
dmLimitLowY　155
dmLimitLoX　657
ヘッダファイル　304
dmLimitLoY　657
ヘッダファイル　304
dmXXXX 定数　657
ヘッダファイル　304
doneBuffer
TEditor のメンバ関数　595
TFileEditor のメンバ関数　610
doSearchReplace
TEditor のメンバ関数　595
doUpdate
TEditor のメンバ関数　595
do_sputn
TTerminal のメンバ関数　542
TTextDevice のメンバ関数　544
dragMode
TView のデータメンバ　149, 155, 546
定数　657
マスク　304
dragView
TView のメンバ関数　553
draw
TAttrSelector のメンバ関数　333
TBackground のメンバ関数　335
TButton のメンバ関数　340
TCheckBoxes のメンバ関数　345
TColorDisplay のメンバ関数　359
TColorSelector のメンバ関数　368
TEditor のメンバ関数　596
TFileInfoPane のメンバ関数　638
TFrame のメンバ関数　389
TGroup のメンバ関数　395
THistory のメンバ関数　408
TIndicator のメンバ関数　614
TInputLine のメンバ関数　420
TLabel のメンバ関数　425
TListViewer のメンバ関数　434
TMenuBar のメンバ関数　439
TMenuBox のメンバ関数　442
TRadioButtons のメンバ関数　486
TScrollBar のメンバ関数　505
TStaticText のメンバ関数　518
TStatusLine のメンバ関数　525
TTerminal のメンバ関数　542
TView のメンバ関数　61, 112, 131, 553
画面のゴミと　64
ビューと　127
クリップ　138, 557
グループと　137
の仕様　67
drawBox
TCluster のメンバ関数　348
DRAWBUF.H　293

drawCursor
TView のメンバ関数 554
drawFlag
TScroller のデータメンバ 508
drawHide
TView のメンバ関数 554
drawLine
TEditor のデータメンバ 588
drawLines
TEditor のメンバ関数 596
drawLock
TScroller のデータメンバ 509
drawPtr
TEditor のデータメンバ 588
drawShow
TView のメンバ関数 554
drawState
TButton のメンバ関数 340
drawSubViews
TGroup のメンバ関数 395
drawUnderRect
TView のメンバ関数 554
drawUnderView
TView のメンバ関数 554
drawView
TView のメンバ関数 61, 555
キャッシュバッファと 673

【E】

editor
TEditWindow のデータメンバ 606
コマンドコード 654
editorDialog
TEditor のデータメンバ 588
editorFlags
TEditor のデータメンバ 589
enableCmd
TCommandSet のメンバ関数 370
enableCommand
TView のメンバ関数 555
enableCommands
TView のメンバ関数 555
endModal
TGroup のメンバ関数 396
TView のメンバ関数 78, 555
endState
TGroup のデータメンバ 393
Enter key
ダイアログボックスと 80
eof
pstream のメンバ関数 328
EOS
定数 658
ヘッダファイル 300
error
pstream のメンバ関数 328
TCollection のメンバ関数 235
TNSCollection のメンバ関数 455
errorAttr
TView のデータメンバ 547
ESC キー
〜の検出 379
evBroadcast 659
ヘッダファイル 296
evCommand 659
ヘッダファイル 296
event データメンバ
イベント定数と 658
eventAvail
TView のメンバ関数 556
eventError 191
TGroup のメンバ関数 396
TView のメンバ関数 176, 177
eventMask
TView のデータメンバ 179, 547
イベント定数と 658
eventQSize 定数 658
evKeyboard 659
ヘッダファイル 296
evKeyDown 659
ヘッダファイル 296
evMessage 659
ヘッダファイル 296
evMouse 176, 659
ヘッダファイル 296
evMouseAuto 659

ヘッダファイル　296
evMouseDown　659
ヘッダファイル　296
evMouseMove　659
ヘッダファイル　296
evMouseUp　659
ヘッダファイル　296
evNothing　659
ヘッダファイル　296
evXXXX 定数　658
execute
TGroup のメンバ関数　175, 396
TMenuView のメンバ関数　445
TView のメンバ関数　556
execView
TGroup のメンバ関数　76, 397
exposed
TView のメンバ関数　556

【F】

fail
pstream のメンバ関数　329
False
Boolean　648
fileList
TFileDialog のデータメンバ　634
fileName
TFileDialog のデータメンバ　634
TFileEditor のデータメンバ　609
FILEVIEW.CPP 例　211
file_block
TFileInfoPane のデータメンバ　637
find
ipstream のメンバ関数　318
opstream のメンバ関数　324
TEditor のメンバ関数　596
findItem
TMenuView のメンバ関数　445
findStr
TEditor のデータメンバ　589
first
TGroup のメンバ関数　397
firstPos
TInputLine のデータメンバ　418
firstThat
TDirCollection のメンバ関数　622
TFileCollection のメンバ関数　630
TGroup のメンバ関数　397
TNSCollection のメンバ関数　220, 221, 455
fixCrtMode
TScreen のメンバ関数　0
flush
opstream のメンバ関数　324
TResourceFile のメンバ関数　496
focusedEvents
イベントクラスと　660
イベント定数と　658
定数　660
ヘッダファイル　306
focusItem
TColorGroupList のメンバ関数　363
TColorItemList のメンバ関数　366
TFileList のメンバ関数　642
TListViewer のメンバ関数　435
focusItemNum
TListViewer のメンバ関数　435
forEach
TCollection のメンバ関数　220
TGroup のメンバ関数　398
TNSCollection のメンバ関数　456
forLabel
TColorDialog のデータメンバ　356
formatLine
TEditor のメンバ関数　596
forSel
TColorDialog のデータメンバ　356
fpbase　312
ヘッダファイル　300
fpstream　242, 314
ヘッダファイル　300
リソースと　258
free　219
TDirCollection のメンバ関数　622
TFileCollection のメンバ関数　630
TNSCollection のメンバ関数　456

freeAll 219
TNSCollection のメンバ関数 456
freeBuffer
TGroup のメンバ関数 399
freeItem
TResourceCollection のメンバ関数 491
TStringCollection のメンバ関数 533

【G】

gapLen
TEditor のデータメンバ 589
GENINC.CPP 660
genRefs
TDrawBuf と 384
TDrawBuffer と 660
TEditor と 605, 660
TEventQueue と 387, 660
TGroup と 405, 660
TTerminal と 543, 660
TView と 569, 660
グローバル関数 660
get
TResourceFile のメンバ関数 497
TStringList のメンバ関数 536
getAltChar グローバル関数 660
getAltCode グローバル関数 661
getBounds
TView のメンバ関数 556
getBuffer
TGroup のメンバ関数 399
getClipRect
TView のメンバ関数 67, 557
getColor
TView のメンバ関数 160, 162, 557
パレットと 162
getCols
TDisplay のメンバ関数 381
getCommands
TView のメンバ関数 557
getCrtMode
TDisplay のメンバ関数 381
getCursorType
TDisplay のメンバ関数 381
getData
TChDirDialog のメンバ関数 619
TCluster のメンバ関数 349
TColorDialog のメンバ関数 357
TGroup のメンバ関数 399
TInputLine のメンバ関数 420
TListBox のメンバ関数 429
TView のメンバ関数 557
getEvent
THWMouse のメンバ関数 416
TMouse のメンバ関数 449
TProgram のメンバ関数 474
TView のメンバ関数 176, 193, 557
～のオーバーライド 194
～の変更 194
getExtent
TView のメンバ関数 129, 558
getFileName
TFileDialog のメンバ関数 635
getHelpCtx
TCluster のメンバ関数 349
TGroup のメンバ関数 399
TMenuView のメンバ関数 446
TView のメンバ関数 201, 558
getItemRect
TMenuBar のメンバ関数 440
TMenuBox のメンバ関数 442
TMenuView のメンバ関数 446
getKey
TSortedListBox のメンバ関数 645
getMouseEvent
TEventQueue のメンバ関数 386
getMousePtr
TEditor のメンバ関数 597
getPalette
TBackground のメンバ関数 335
TButton のメンバ関数 340
TCluster のメンバ関数 349
TDialog のメンバ関数 378
TEditor のメンバ関数 597
TFileInfoPane のメンバ関数 638
TFrame のメンバ関数 389

THistory のメンバ関数　408
THistoryViewer のメンバ関数　411
THistoryWindow のメンバ関数　414
TIndicator のメンバ関数　614
TInputLine のメンバ関数　421
TLabel のメンバ関数　425
TListViewer のメンバ関数　435
TMemo のメンバ関数　616
TMenuView のメンバ関数　446
TProgram のメンバ関数　475
TScrollBar のメンバ関数　505
TScroller のメンバ関数　510
TStaticText のメンバ関数　518
TStatusLine のメンバ関数　525
TView のメンバ関数　163, 558
TWindow のメンバ関数　575
～のオーバーライド　163
getRows
TDisplay のメンバ関数　381
getSelection
THistoryWindow のメンバ関数　414
getState
TView のメンバ関数　558
getText
TColorGroupList のメンバ関数　363
TColorItemList のメンバ関数　367
TDirListBox のメンバ関数　626
TFileList のメンバ関数　642
THistoryViewer のメンバ関数　411
TListBox のメンバ関数　429
TListViewer のメンバ関数　435
TParamText のメンバ関数　466
getTitle
TEditWindow のメンバ関数　607
TWindow のメンバ関数　575
gfGrowAll　154, 662
ヘッダファイル　304
gfGrowHiX　154, 662
ヘッダファイル　304
gfGrowHiY　154, 662
ヘッダファイル　304
gfGrowLoX　153, 662
ヘッダファイル　304
gfGrowLoY　154, 662
ヘッダファイル　304
gfGrowRel　154, 662
ヘッダファイル　304
gfXXXX 定数　661
ヘッダファイル　304
good
pstream のメンバ関数　329
grow
TRect のメンバ関数　129, 489
growMode
TView のデータメンバ　149, 153, 547
定数　661
マスク　304, 661
growTo
TView のメンバ関数　558

【H】

handleEvent　→イベント，処理
TAttrSelector のメンバ関数　333
TButton のメンバ関数　340
TChDirDialog のメンバ関数　619
TCluster のメンバ関数　349
TColorDialog のメンバ関数　357
TColorDisplay のメンバ関数　359
TColorItemList のメンバ関数　367
TColorSelector のメンバ関数　369
TDeskTop のメンバ関数　375
TDialog のメンバ関数　378
TDirListBox のメンバ関数　626
TEditor のメンバ関数　597
TEditWindow のメンバ関数　607
TFileDialog のメンバ関数　635
TFileEditor のメンバ関数　610
TFileInfoPane のメンバ関数　638
TFileInputLine のメンバ関数　639
TFileList のメンバ関数　643
TFrame のメンバ関数　389
TGroup のメンバ関数　400
THistory のメンバ関数　409
THistoryViewer のメンバ関数　412
TInputLine のメンバ関数　421

TLabel のメンバ関数　425
TListViewer のメンバ関数　435
TMemo のメンバ関数　616
TMenuView のメンバ関数　446
TProgram のメンバ関数　475
TScrollBar のメンバ関数　505
TScroller のメンバ関数　511
TSortedListBox のメンバ関数　645
TStatusLine のメンバ関数　525
TView のメンバ関数　132, 176, 187, 559
TWindow のメンバ関数　575
～のオーバーライド　187
～の概要　187
～の継承　187
～の呼び出し　201
has
TCommandSet のメンバ関数　370
hasSelection
TEditor のメンバ関数　597
hcDragging　662
ヘッダファイル　305
hcNoContext　662
ヘッダファイル　305
hcNoContext 定数　49, 201
hcXXXX 定数　662
HELLO.CPP　22～34
main()　32
run メンバ関数　34
コンストラクタ　32
helpCtx
TView のデータメンバ　201, 548
HELPME!.DOC ファイル　5, 6
hide
THWMouse のメンバ関数　416
TMouse のメンバ関数　449
TView のメンバ関数　559
hideCursor
TView のメンバ関数　559
hideSelect
TEditor のメンバ関数　597
hint
TStatusLine のメンバ関数　526
historyAdd グローバル関数　663
historyCount グローバル関数　663
historyID
THistory のデータメンバ　407
THistoryViewer のデータメンバ　411
historyStr グローバル関数　663
historyWidth
THistoryViewer のメンバ関数　412
hotKey
TMenuView のメンバ関数　447
hScrollBar
TEditor のデータメンバ　589
TListViewer のデータメンバ　432
TScroller のデータメンバ　509

【I】

ID 番号
ヒストリーリスト　407
idle
TApplication のメンバ関数　194, 195
TProgram のメンバ関数　476
時間
アプリケーション　476
～の使用　195
ifpstream　242, 315
ヘッダファイル　300
index
TColorItem のデータメンバ　364
TResourceFile のデータメンバ　494
indexOf
TDirCollection のメンバ関数　623
TFileCollection のメンバ関数　630
TGroup のメンバ関数　401
TNSCollection のメンバ関数　219, 457
indexPos
TResourceFile のデータメンバ　494
indicator
TEditor のデータメンバ　589
init
pstream のメンバ関数　329
initBackground
TDeskTop のメンバ関数　375

initBuffer
TEditor のメンバ関数　597
TFileEditor のメンバ関数　610
initDeskTop　43
TProgram のメンバ関数　41, 476
inited
TVMemMgr のデータメンバ　570
initFrame
TWindow のメンバ関数　576
initHistoryWindow
THistory のメンバ関数　409
initMenuBar
TProgram のメンバ関数　41, 476
initScreen
TProgram のメンバ関数　477
initStatusLine
TProgram のメンバ関数　41, 477
initTypes
pstream のメンバ関数　329
initViewer
THistoryWindow のメンバ関数　414
inputBox グローバル関数　663
ヘッダファイル　294
inputBoxRect グローバル関数　664
ヘッダファイル　294
insCount
TEditor のデータメンバ　590
insert
TDirCollection のメンバ関数　623
TFileCollection のメンバ関数　630
TGroup のメンバ関数　56, 134, 401
TNSCollection のメンバ関数　457
TNSSortedCollection のデータメンバ　227
TNSSortedCollection のメンバ関数　461
insertBefore
TGroup のメンバ関数　401
insertBuffer
TEditor のメンバ関数　598
insertFrom
TEditor のメンバ関数　598
insertText
TEditor のメンバ関数　598
Int11trap
ヘッダファイル　296
intermediary オブジェクト　196
intersect
TRect のメンバ関数　489
I/O →ストリーム
基本的な操作　312
ストリーム　314, 315, 316, 317, 321
iopstream　242, 316
ヘッダファイル　300
ipstream　242, 317
TPReadObjects と　470
TStreamableClass と　531
～のフレンド　319
ヘッダファイル　300
isClipboard
TEditor のメンバ関数　599
isEmpty
TCommandSet のメンバ関数　370
TRect のメンバ関数　489
isSelected
TDirListBox のメンバ関数　626
TListViewer のメンバ関数　436
isValid
TEditor のデータメンバ　590

【K】

kbXXXX 定数　664
keyAt
TResourceFile のメンバ関数　497
keyCode
TStatusItem のデータメンバ　521
KeyDownEvent 構造体　320
keyEvent
TView のメンバ関数　559
KeyEventDown　190
keyOf
TResourceCollection のメンバ関数　492
keyState
TEditor のデータメンバ　590

【L】

last
　TGroup のデータメンバ　393
lastThat
　TDirCollection のメンバ関数　623
　TFileCollection のメンバ関数　630
　TNSCollection のメンバ関数　220，221，457
light
　TLabel のデータメンバ　424
limit
　TCollection のデータメンバ　216
　TEditor のデータメンバ　590
　TNSCollection のデータメンバ　453
　TScroller のデータメンバ　509
lineEnd
　TEditor のメンバ関数　599
lineMove
　TEditor のメンバ関数　599
lineStart
　TEditor のメンバ関数　599
link
　THistory のデータメンバ　407
　TLabel のデータメンバ　424
__link マクロ　262，302
　ヘッダファイル　300
　例　121
list
　TDirListBox のメンバ関数　627
　TFileList のメンバ関数　643
　TListBox のメンバ関数　430
　TSortedListBox のメンバ関数　646
loadFile
　TFileEditor のメンバ関数　610
locate
　TView のメンバ関数　560
location
　TIndicator のデータメンバ　613
lock
　TEditor のメンバ関数　599
　TGroup のメンバ関数　401
lockCount
　TEditor のデータメンバ　590
lockFlag
　TGroup のデータメンバ　393
lookup
　TStreamableTypes のメンバ関数　532
lowMemory グローバル関数　204，667

【M】

makeDefault
　TButton のメンバ関数　341
makeFirst
　TView のメンバ関数　560
makeGlobal
　TView のメンバ関数　560
makeLocal
　TView のメンバ関数　560
mapColor
　TView のメンバ関数　560
mark
　TAttrSelector のメンバ関数　333
　TCheckBoxes のメンバ関数　345
　TCluster のメンバ関数　349
　TRadioButtons のメンバ関数　486
matches
　TGroup のメンバ関数　402
max
　TStatusDef のデータメンバ　520
maxCollectionSize 変数　235，667
maxFindStrLen 定数　667
maxLen
　TInputLine のデータメンバ　418
maxReplaceStrLen 定数　668
maxVal
　TScrollBar のデータメンバ　502
maxViewWidth 定数　668
mbLeftButton　668
　ヘッダファイル　296
mbRightButton　668
　ヘッダファイル　296
mbXXXX 定数　668
menuBar
　TProgram のデータメンバ　48，472

ポインタ　42
MENUS.H　293
message
グローバル関数　669
messageBox グローバル関数　669
定数　670
ヘッダファイル　294
messageBoxRect グローバル関数　670
ヘッダファイル　294
MessageEvent
ヘッダファイル　296
MessageEvent 構造体　320
mfCancelButton　670
ヘッダファイル　294
mfConfirmation　670
ヘッダファイル　294
mfError　670
ヘッダファイル　294
mfInformation　670
ヘッダファイル　294
mfNoButton　670
ヘッダファイル　294
mfOKButton　670
ヘッダファイル　294
mfOKCancel　671
ヘッダファイル　294
mfWarning　670
ヘッダファイル　294
mfYesButton　670
ヘッダファイル　294
mfYesNoCancel　671
ヘッダファイル　294
min
TStatusDef のデータメンバ　520
minVal
TScrollBar のデータメンバ　503
modified
TEditor のデータメンバ　590
TIndicator のデータメンバ　613
TResourceFile のデータメンバ　494
monoSeg
ヘッダファイル　296
mouseEvent
TView のメンバ関数　561
MouseEventType　190
ヘッダファイル　296
mouseInView
TView のメンバ関数　561
move
TRect のメンバ関数　489
moveBuf
TDrawBuffer のメンバ関数　383
moveChar
TDrawBuf のメンバ関数　65
TDrawBuffer のメンバ関数　383
moveCStr
TDrawBuffer のメンバ関数　383
movedTo
TAttrSelector のメンバ関数　333
TCluster のメンバ関数　350
TRadioButtons のメンバ関数　486
moveStr
TDrawBuf のメンバ関数　65
TDrawBuffer のメンバ関数　383
moveTo
TView のメンバ関数　562
MSGBOX.H　294

【N】

name
TColorGroup のデータメンバ　361
TColorItem のデータメンバ　364
TResourceCollection のデータメンバ　491
new 演算子　674
destroy と　79
セーフティプールと　205
newColor
TAttrSelector のメンバ関数　333
newDirectory
TDirListBox のメンバ関数　627
newLine
TEditor のメンバ関数　599
newList
TFileList のメンバ関数　643
TListBox のメンバ関数　430

TSortedListBox のメンバ関数　646
newStr グローバル関数　228, 671
next
TColorGroup のデータメンバ　361
TColorItem のデータメンバ　364
TSItem のデータメンバ　513
TStatusDef のデータメンバ　520
TStatusItem のデータメンバ　521
TView のデータメンバ　548
nextChar
TEditor のメンバ関数　600
nextLine
TEditor のメンバ関数　600
TTerminal のメンバ関数　542
nextView
TView のメンバ関数　562
nextWord
TEditor のメンバ関数　600
No ボタン
コマンド　653
normalCursor
TView のメンバ関数　562
number
TWindow のデータメンバ　573
numCols
TListViewer のデータメンバ　432

【O】

OBJECTS.H　295
ofBuffered　151, 673
ヘッダファイル　304
ofCentered　153, 672-673
ヘッダファイル　304
ofCenterX　153, 673
ヘッダファイル　304
ofCenterY　153, 673
ヘッダファイル　304
ofFirstClick　150, 672
ヘッダファイル　304
ofFramed　151, 672
ヘッダファイル　304
ofPostProcess　151, 673
ヘッダファイル　304
ofPreProcess　151, 673
ヘッダファイル　304
ofpstream　242, 321
ヘッダファイル　300
ofSelectable　150, 672
ヘッダファイル　304
ofTileable　152, 673
ヘッダファイル　304
ofTopSelect　150, 672
ヘッダファイル　304
ofXXXX 定数　672, →フラグ, options
ヘッダファイル　304
Ok ボタン
コマンド　653
open
fpbase のメンバ関数　313
fpstream のメンバ関数　315
ifpstream のメンバ関数　316
ofpstream のメンバ関数　322
opstream　241, 323
TStreamableClass と　531
～のフレンド　326
ヘッダファイル　300
options
TEditor　584
TView のデータメンバ　149, 548
キーの局所化と　90
ヘッダファイル　304～305
フラグ　672
origin
TView のデータメンバ　128, 548
outOfMemory
TApplication のメンバ関数　207
TProgram のメンバ関数　477
overflow
TTextDevice のメンバ関数　544
overwrite
TEditor のデータメンバ　590
owner
TView のデータメンバ　548

【P】

pack

TNSCollection のメンバ関数　458
pal
TColorDialog のデータメンバ　356
palette
TWindow のデータメンバ　573
paramCount
TParamText のデータメンバ　465
paramList
TParamText のデータメンバ　465
parentMenu
TMenuView のデータメンバ　444
pattern
TBackground のデータメンバ　334
pgStep
TScrollBar のデータメンバ　503
phaseType
列挙型　394
pos
TResourceItem のデータメンバ　261
positionalEvents
イベントクラスと　660
イベント定数と　658
定数　675
ヘッダファイル　306
present
THWMouse のメンバ関数　416
TMouse のメンバ関数　450
press
TAttrSelector のメンバ関数　333
TButton のメンバ関数　341
TCheckBoxes のメンバ関数　345
TCluster のメンバ関数　350
TRadioButtons のメンバ関数　486
prev
TView のメンバ関数　562
prevChar
TEditor のメンバ関数　600
prevLine
TEditor のメンバ関数　600
prevLines
TTerminal のメンバ関数　542
prevView
TView のメンバ関数　562
prevWord
TEditor のメンバ関数　600
pstream　327
ストリーム入出力可能なオブジェクトと　238
～のフレンド　330
ヘッダファイル　300
put
TResourceFile のメンバ関数　497
TStrListMaker のメンバ関数　538
putAttribute
TDrawBuffer のメンバ関数　384
putChar
TDrawBuffer のメンバ関数　384
putEvent
TProgram のメンバ関数　478
TView のメンバ関数　563
putInFrontOf
TView のメンバ関数　563

【Q】

queBack
TTerminal のデータメンバ　540
queEmpty
TTerminal のメンバ関数　542
queFront
TTerminal のデータメンバ　540
queues
イベント
～の作成　386
～の消去　386

【R】

range
TListViewer のデータメンバ　432
rdbuf
fpbase のメンバ関数　313
fpstream のメンバ関数　315
ifpstream のメンバ関数　316
ofpstream のメンバ関数　322
pstream のメンバ関数　329
rdstate
pstream のメンバ関数　330

read
TBackground のメンバ関数 335
TButton のメンバ関数 342
TChDirDialog のメンバ関数 620
TCluster のメンバ関数 350
TCollection のメンバ関数 353
TColorDialog のメンバ関数 357
TColorDisplay のメンバ関数 360
TColorGroupList のメンバ関数 363
TColorSelector のメンバ関数 369
TDirCollection のメンバ関数 623
TDirListBox のメンバ関数 627
TEditor のメンバ関数 600
TEditWindow のメンバ関数 607
TFileCollection のメンバ関数 631
TFileDialog のメンバ関数 635
TFileEditor のメンバ関数 610
TFileInfoPane のメンバ関数 638
TFileInputLine のメンバ関数 640
TFileList のメンバ関数 643
TGroup のメンバ関数 402
THistory のメンバ関数 409
TIndicator のメンバ関数 614
TInputLine のメンバ関数 421
TLabel のメンバ関数 425
TListBox のメンバ関数 430
TListViewer のメンバ関数 436
TMemo のメンバ関数 616
TMenuBox のメンバ関数 442
TMenuView のメンバ関数 447
TParamText のメンバ関数 466
TResourceCollection のメンバ関数 492
TScrollBar のメンバ関数 505
TScroller のメンバ関数 511
TSortedCollection のメンバ関数 515
TStaticText のメンバ関数 518
TStatusLine のメンバ関数 526
TStreamable のメンバ関数 528
TStringCollection のメンバ関数 533
TStringList のメンバ関数 536
TView のメンバ関数 564
TWindow のメンバ関数 576
ストリーム入出力可能なクラスへのリンク 251
独自の～関数の作成 254
readByte
ipstream のメンバ関数 318
readBytes
ipstream のメンバ関数 318
readData
ipstream のメンバ関数 318
readDirectory
TFileList のメンバ関数 643
readItem
TCollection のメンバ関数 353
TDirCollection のメンバ関数 623
TFileCollection のメンバ関数 631
TResourceCollection のメンバ関数 492
TSortedCollection のメンバ関数 515
TStringCollection のメンバ関数 533
README.DOC 5
readPrefix
ipstream のメンバ関数 318
readString
ipstream のメンバ関数 318
readSuffix
ipstream のメンバ関数 318
readWord
ipstream のメンバ関数 318
redraw
TGroup のメンバ関数 402
registerHandler
THWMouse のメンバ関数 416
TMouse のメンバ関数 450
registerObject
ipstream のメンバ関数 319
opstream のメンバ関数 324
registerTypes
TStreamableTypes のメンバ関数 532
remove
TDirCollection のメンバ関数 624
TFileCollection のメンバ関数 631
TGroup のメンバ関数 402

TNSCollection のメンバ関数　219, 458
TResourceFile のメンバ関数　497
removeAll
TNSCollection のメンバ関数　219, 458
removeView
TGroup のメンバ関数　402
replace
TEditor のメンバ関数　600
replaceStr
TEditor のデータメンバ　590
resetCurrent
TGroup のメンバ関数　403
resetCursor
TView のメンバ関数　564
RESOURCE.H　295
resume
TEventQueue　386
THWMouse のメンバ関数　416
TMouse のメンバ関数　450
TScreen のメンバ関数　501
TSystemError のメンバ関数　540
run
TProgram のメンバ関数　478

【S】

safetyPool
TVMemMgr のデータメンバ　570
safetyPoolExhausted
lowMemory 関数と　667
safetyPoolSize
TVMemMgr のデータメンバ　570
save
TFileEditor のメンバ関数　611
saveAs
TFileEditor のメンバ関数　611
saveFile
TFileEditor のメンバ関数　611
sbDownArrow　676
ヘッダファイル　305
sbHandleKeyboard　677
ヘッダファイル　305
sbHorizontal　677
ヘッダファイル　305
sbIndicator　676
ヘッダファイル　305
sbLeftArrow　676
ヘッダファイル　305
sbPageDown　676
ヘッダファイル　305
sbPageLeft　676
ヘッダファイル　305
sbPageRight　676
ヘッダファイル　305
sbPageUp　676
ヘッダファイル　305
sbRightArrow　676
ヘッダファイル　305
sbUpArrow　676
ヘッダファイル　305
sbVertical　677
ヘッダファイル　305
sbXXXX 定数　676
ヘッダファイル　305
screenHeight
TScreen のデータメンバ　499
screenMode
TScreen のデータメンバ　499
screenWidth
TScreen のデータメンバ　499
scrollDraw
TScrollBar のメンバ関数　505
TScroller のメンバ関数　511
scrollStep
TScrollBar のメンバ関数　506
scrollTo
TEditor のメンバ関数　601
TScroller のメンバ関数　511
search
TEditor のメンバ関数　601
TNSSortedCollection のメンバ関数　227, 461
seekg
ipstream のメンバ関数　319
seekp

opstream のメンバ関数　324
sel
TCluster のデータメンバ　347
select　→フォーカス，ビュー
options のデータメンバと　150，672
TView のメンバ関数　145，564
TWindow のデータメンバ
メッセージ　306
selectAll
TInputLine のメンバ関数　421
selecting
TEditor のデータメンバ　591
selectItem
TListViewer のメンバ関数　436
selectMode
列挙型　677
selectNext
TGroup のメンバ関数　403
selEnd
TEditor のデータメンバ　591
TInputLine のデータメンバ　419
selStart
TEditor のデータメンバ　591
TInputLine のデータメンバ　419
selType
TColorSelector のデータメンバ　368
setBounds
TView のメンバ関数　564
setbuf
fpbase のメンバ関数　313
setBufLen
TEditor のメンバ関数　602
setBufSize
TEditor のメンバ関数　602
TFileEditor のメンバ関数　611
setCmdState
TEditor のメンバ関数　602
setColor
TColorDisplay のメンバ関数　360
setCommands
TView のメンバ関数　564
setCrtData
TScreen のメンバ関数　501
setCrtMode
TDisplay のメンバ関数　381
setCurPtr
TEditor のメンバ関数　602
setCurrent
TGroup のメンバ関数　403
setCursor
TView のメンバ関数　565
setCursorType
TDisplay のメンバ関数　382
setData
TChDirDialog のメンバ関数　620
TCluster のメンバ関数　350
TColorDialog のメンバ関数　358
TGroup のメンバ関数　404
TInputLine のメンバ関数　422
TListBox のメンバ関数　430
TParamText のメンバ関数　466
TRadioButtons のメンバ関数　487
TView のメンバ関数　86，565
setLimit
TNSCollection のメンバ関数　458
TScroller のメンバ関数　511
setParams
TScrollBar のメンバ関数　506
setRange
THWMouse のメンバ関数　416
TListViewer のメンバ関数　436
TMouse のメンバ関数　450
TScrollBar のメンバ関数　506
setScreenMode
TProgram のメンバ関数　478
setSelect
TEditor のメンバ関数　603
setstate
pstream のメンバ関数　330
setState
TButton のメンバ関数　342
TCluster のメンバ関数　350
TDirListBox のメンバ関数　627
TEditor のメンバ関数　603
TFrame のメンバ関数　389
TGroup のメンバ関数　404

TIndicator のメンバ関数　614
TInputLine のメンバ関数　422
TListViewer のメンバ関数　437
TScroller のメンバ関数　512
TView のメンバ関数　157, 565
TWindow のメンバ関数　576
～のオーバーライド　158
setStep
TScrollBar のメンバ関数　507
setValue
TIndicator のメンバ関数　614
TScrollBar のメンバ関数　507
setVideoMode
TScreen のメンバ関数　501
sfActive　678
ヘッダファイル　304
sfCursorIns　678
ヘッダファイル　304
sfCursorVis　678
ヘッダファイル　304
sfDefault　679
ヘッダファイル　304
sfDisabled　678
ヘッダファイル　304
sfDragging　678
ヘッダファイル　304
sfExposed　679
ヘッダファイル　304
sfFocused　678
ヘッダファイル　304
sfModal　679
ヘッダファイル　304
sfSelected　678
ヘッダファイル　304
sfVisible　678
ヘッダファイル　304
sfVShadow　678
ヘッダファイル　304
sfXXXX 定数　677, →フラグ, state
ヘッダファイル　304
shouldDelete
TNSCollection のデータメンバ　219, 453
show
THWMouse のメンバ関数　417
TMouse のメンバ関数　450
TView のメンバ関数　566
showCursor
TView のメンバ関数　566
showMarkers
specialChars と　680
TView のデータメンバ　549
shutDown
destroy と　207, 463
TChDirDialog のメンバ関数　620
TDeskTop のメンバ関数　375
TEditor のメンバ関数　603
TFileDialog のメンバ関数　635
TGroup のメンバ関数　404
THistory のメンバ関数　409
TLabel のメンバ関数　426
TListViewer のメンバ関数　437
TObject のメンバ関数　463
TProgram のメンバ関数　479
TScroller のメンバ関数　512
TView のメンバ関数　549
TWindow のメンバ関数　576
size
TResourceItem のデータメンバ　261
TView のデータメンバ　128, 549
sizeLimits
TView のメンバ関数　567
TWindow のメンバ関数　72, 577
specialChars
showMarkers と　680
ヘッダファイル　300
変数　680
standardScrollBar
TWindow のメンバ関数　577
startSelect
TEditor のメンバ関数　603
startupCursor
TScreen のデータメンバ　500
startupMode
TScreen のデータメンバ　500
state

pstream のデータメンバ　327
TView のデータメンバ　149, 156, 549
～の設定　156
フラグ　677
ヘッダファイル　304
フラグ　558
statusLine
TProgram のデータメンバ　473
イベントと　184
～へのポインタ　42
StreamableInit
ヘッダファイル　300
列挙型　680
streamableName
TStreamable のメンバ関数　528
ストリーム入出力可能なクラスと　250
suspend
TEventQueue　387
THWMouse のメンバ関数　417
TMouse のメンバ関数　450
TScreen のメンバ関数　501
TSystemError のメンバ関数　540
SYSTEM.H　296

【T】

TAB の順序　80, 81, 146, →Z オーダー
TAB キー
ダイアログボックスと　26, 80
フォーカスコントロールと　145～146
TApplication　28, 38, 114, 330, →アプリケーション
TProgram 対　330
ヘッダファイル　291
例　41
TAttrSelector　332
TBackground　138, 334, →バックグラウンド
ヘッダファイル　291
TBufListEntry　337
セーフティプールと　205
ヘッダファイル　291
TButton　115, 337, →ボタン
ヘッダファイル　292
TChDirDialog　617
コマンド　618
TCheckBoxes　343, →チェックボックス
ヘッダファイル　292
例　74
TCluster　116, 346, →クラスタ
ヘッダファイル　292
例　74
TCollection　122, 215, 352, →コレクション
ストリームと　244
TCollectionName
TCollection のデータメンバ　352
TColorDialog　354
TColorDisplay　358
TColorGroup　360
TColorGroupList　361
TColorItem　364
TColorItemList　365
TColorSelector　367
TCommandSet　369
演算子　159
TDeskInit　372
ヘッダファイル　291
TDeskTop　113, 373, →デスクトップ
ヘッダファイル　291
例　41
TDialog　115, 377, →ダイアログボックス
～の性質　377
標準コマンド
ヘッダファイル　304
ヘッダファイル　292
例　74
TDirCollection　621
TDirEntry　625
TDirListBox　625
TDisplay
ヘッダファイル　296
TDisplay クラス　380
TDrawBuf
～のフレンド　384
TDrawBuffer　64, 382

genRefs と 660
のメンバ関数 66
TEditor 582
genRefs と 660
エディタフラグ 586
オプション 584
キー割り当て 585
検索文字列の長さ 667
コマンド定数 586
ダイアログ定数 586
置換文字列の長さ 668
定数 585
～のフレンド 605
バッファギャップ 582
パレット 586
ブロックコマンド 585
ブロックの選択 584
TEditWindow 606
tellg
ipstream のメンバ関数 319
tellp
opstream のメンバ関数 324
terminal ビュー 115, 133
イベントと 177
TEvent 169, 385, →イベントオブジェクト
what データメンバ 171
定義 189
ヘッダファイル 296
TEventQueue 386
genRefs と 660
TMouse と 387
～のフレンド 387
ヘッダファイル 296
TEXTVIEW.H 296
TFileCollection 628
TFileDialog 632
オプション 632
コマンド 633
TFileEditor 609
TFileInfoPane 637
TFileInputLine 639
TFileList 641
コマンド 641
TFrame 115, 141, 388, →フレーム
TGroup 113, 391, →グループ
genRefs と 660
データメンバ 113
～のフレンド 405
例 41, 53, 74
THistInit 406
ヘッダファイル 292
THistory 117, 407, →ヒストリーリスト
ヘッダファイル 292
THistoryViewer 410, →ヒストリーリスト, ビュアー
ヘッダファイル 292
THistoryWindow 413, →ヒストリーリスト, ウィンドウ
ヘッダファイル 292
THWMouse 415
ヘッダファイル 296
tile
TDeskTop のメンバ関数 375
tileError
TDeskTop のメンバ関数 375
TIndicator 613
TInputLine 117, 417, →入力行
ヘッダファイル 292
例 74
title
TButton のデータメンバ 339
TWindow のデータメンバ 573
TKEYS.H 297
TLabel 423, →ラベル
オブジェクト
チェックボックス と 343
クラスタと 346
ヘッダファイル 292
例 74
TListBox 118, 427, →リストボックス
ヘッダファイル 292
TListViewer 118, 431
メッセージ 306
TMemo 615
TMemoRec 615

TMenuBar　438，→メニュー
　ヘッダファイル　293
　例　41，48，49
TMenuBox　441，→メニュー
　ヘッダファイル　293
　例　41
TMenuItem　→メニュー
　＋演算子　674
　ヘッダファイル　293
　例　49
TMenuView　116，443，→メニュー
　ヘッダファイル　293
TMouse　449
　TEventQueue と　387
　ヘッダファイル　296
TMouseEventType　451
TNSCollection　215，451
　ヘッダファイル　303
TNSSortedCollection　224，459
　ヘッダファイル　303
TObject　111，462
　導出クラス
　　～のインスタンスの消去　207，462
　ヘッダファイル　303
TOBJSTRM.H　300
toggleInsMode
　TEditor のメンバ関数　603
topItem
　TListViewer のデータメンバ　433
topView
　TView のメンバ関数　567
TPalette　160～165，463
TParamText　464
　ヘッダファイル　292
TPoint　110，128，467
　～のフレンド　468
TPReadObjects　469
　～のフレンド　470
　ヘッダファイル　300
TProgInit　470
　ヘッダファイル　291
　例　41
TProgram　114，471，→アプリケーション
　TApplication 対　330
　ヘッダファイル　291
　例　41
TPWObj　483
　～のフレンド　484
　ヘッダファイル　300
TPWrittenObjects　484
　～のフレンド　485
　ヘッダファイル　300
trackCursor
　TEditor のメンバ関数　604
TRadioButtons　485，→ラジオボタン
　ヘッダファイル　292
　例　74
TRect　110，129，487
　例　53
TResourceCollection　123，257，490，
　→コレクション/リソース
　ヘッダファイル　295
　リソースと　257
TResourceFile　123，257，493，
　→リソース
　構成要素　261
　ヘッダファイル　295
　リソースと　257
TResourceItem　490
　TStrIndexRec 対　267
　ヘッダファイル　295
True
　Boolean　648
TScreen　499
　ヘッダファイル　296
TScrollBar　118，141，502，
　→スクロールバー
　TScrollChars typedef と　680
　メッセージ　306
　例　53
TScrollChars typedef　680
　ヘッダファイル　306
TScroller　118，141，508，→スクローラ
　例　53
TSearchRec　628，637
TSItem　513

ヘッダファイル　292
TSortedCollection　122，224，514，
→コレクション，ソート済み
TSortedListBox　645
TStaticText　119，516，→テキスト，静的
ヘッダファイル　292
TStatusDef　519
＋演算子　674
ヘッダファイル　293
TStatusDef コンストラクタ
ヘルプコンテキストと　202
TStatusItem　520
＋演算子　674
ヘッダファイル　293
TStatusLine　120，523，
→ステータスライン
ヘッダファイル　293
例　41
TStreamable　215，527
ストリームマネージャと　245
～のフレンド　529
ヘッダファイル　300
TStreamableClass　529
クラス　121
ストリームマネージャと　245
～のフレンド　531
ヘッダファイル　300
TStreamableTypes　531
TStreamableClass と　531
ストリームマネージャと　245
ヘッダファイル　300
TStrIndexRec　535
TResourceItem 対　267
ヘッダファイル　295
TStringCollection　123，532，
→コレクション，文字列
TStringList 対　267
ヘッダファイル　295
TStringList　124，535，→文字列リスト
TStringCollection 対　267
TStrListMaker 対　269
ヘッダファイル　295
TStrListMaker　537，→文字列リスト
TStringList 対　269
ヘッダファイル　295
TSubMenu
＋演算子　674
ヘッダファイル　293
TSystemError　539
TDrawBuf と　384
ヘッダファイル　296
TTerminal　119，540，
→テキスト，デバイス
genRefs と　660
～のフレンド　543
ヘッダファイル　296
TTextDevice　119，543，
→テキスト，デバイス
ヘッダファイル　296
TTYPES.H　300
Turbo Vision
アプリケーション設計　276
オブジェクト階層　127
活用法　15
クラスの概要　102
継承　15
座標系　128，130
名前の付け方　287
～の拡張　15
～の定義　14
～のデバッグ　271
～の要素　17
～へのアプリケーションの移植　275
TV.H
アプリケーション　262
インクルードの制御　301
TV.LIB
genRefs と　660
TVideoBuf
ヘッダファイル　291
TView　545，→ビュー
genRefs と　660
state マスク
ヘッダファイル　304
TDrawBuf と　384
TDrawBuffer のフレンドクラス　66

TEventQueue と 387
～のフレンド 569
例 41, 53, 74
TVMemMgr 569
セーフティプールと 205
ヘッダファイル 291
TVOBJS.H 303
TVWRITE.INC
TV.LIB と 660
TWindow 114, 572, →ウィンドウ
データメンバ 114
～の基本クラス 572
パレット 305
Twindow
番号定数 305
TWindow
メッセージ 306
例 53
TWindowInit 579
TWindow と 572
typecasting
コレクション と 225

【U】

uchar
typedef 681
ヘッダファイル 300
undo
TEditor 587, 588, 590
TEditor のメンバ関数 604
Union
TRect のメンバ関数 489
unlock
TEditor のメンバ関数 604
TGroup のメンバ関数 405
update
TEditor のメンバ関数 604
TStatusLine のメンバ関数 526
updateCommands
TEditor のメンバ関数 604
TFileEditor のメンバ関数 612
updateFlags
TEditor のデータメンバ 591
Uses_TClassName 識別子 301
～の定義 262
ushort
typedef 681
ヘッダファイル 300

【V】

valid
TChDirDialog のメンバ関数 620
TDialog のメンバ関数 379
TEditor のメンバ関数 604
TFileDialog のメンバ関数 635
TFileEditor のメンバ関数 612
TGroup のメンバ関数 405
TView のメンバ関数 208, 210, 567
～のオーバーライド 208
例 208
validView
TApplication のメンバ関数 207
TProgram のメンバ関数 479
セーフティプールと 206
videoModes
TDisplay のメンバ関数 382
viewer
THistoryWindow のデータメンバ 413
VIEWS.H 303
virtual メンバ関数 →メンバ関数
void *()
pstream のメンバ関数 329
void *() 演算子
pstream のメンバ関数 329
vScrollBar
TEditor のデータメンバ 591
TListViewer のデータメンバ 433
TScroller のデータメンバ 509

【W】

wfClose 682
ヘッダファイル 305
wfGrow 682
ヘッダファイル 305
wfMove 682

ヘッダファイル　305
wfXXXX 定数　681
wfZoom　682
ヘッダファイル　305
what
TEvent のデータメンバ　171, 385
wildCard
TFileDialog のデータメンバ　634
wnNoNumber　682
ヘッダファイル　305
wpBlueWindow　683
ヘッダファイル　305
wpCyanWindow　683
ヘッダファイル　305
wpGrayWindow　683
ヘッダファイル　305
wpXXXX 定数　683
write
TBackground のメンバ関数　335
TButton のメンバ関数　342
TChDirDialog のメンバ関数　621
TCluster のメンバ関数　351
TCollection のメンバ関数　354
TColorDialog のメンバ関数　358
TColorDisplay のメンバ関数　360
TColorGroupList のメンバ関数　363
TColorSelector のメンバ関数　369
TDirCollection のメンバ関数　624
TDirListBox のメンバ関数　627
TEditor のメンバ関数　605
TEditWindow のメンバ関数　607
TFileCollection のメンバ関数　631
TFileDialog のメンバ関数　636
TFileEditor のメンバ関数　612
TFileInfoPane のメンバ関数　638
TFileInputLine のメンバ関数　640
TFileList のメンバ関数　644
TGroup のメンバ関数　405
THistory のメンバ関数　409
TIndicator のメンバ関数　614
TInputLine のメンバ関数　422
TLabel のメンバ関数　426
TListBox のメンバ関数　430
TListViewer のメンバ関数　437
TMemo のメンバ関数　616
TMenuBox のメンバ関数　442
TMenuView のメンバ関数　447
TParamText のメンバ関数　466
TResourceCollection のメンバ関数 492
TScrollBar のメンバ関数　507
TScroller のメンバ関数　512
TSortedCollection のメンバ関数　516
TStaticText のメンバ関数　518
TStatusLine のメンバ関数　526
TStreamable のメンバ関数　529
TStringCollection のメンバ関数　534
TStringList のメンバ関数　536
TStrListMaker のメンバ関数　538
TView のメンバ関数　568
TWindow のメンバ関数　577
ストリーム入出力可能なクラスへのリンク　251
独自の～関数の作成　254
write_args 構造体　683
writeBuf
TView のメンバ関数　65, 67, 568
writeByte
opstream のメンバ関数　325
writeBytes
opstream のメンバ関数　325
writeChar
TView のメンバ関数　62, 568
writeData
opstream のメンバ関数　325
writeItem
TDirCollection のメンバ関数　624
TFileCollection のメンバ関数　631
TResourceCollection のメンバ関数 492
TSortedCollection のメンバ関数　516
TStringCollection のメンバ関数　534
writeLine
TView のメンバ関数　568, 65, 66
writePrefix
opstream のメンバ関数　325

writeStr
TView のメンバ関数 62, 569
writeString
opstream のメンバ関数 325
writeSuffix
opstream のメンバ関数 325
writeWord
opstream のメンバ関数 326

【X】

x
TPoint のデータメンバ 128, 467

【Y】

y
TPoint のデータメンバ 128, 467
Yes ボタン
コマンド 653

【Z】

Z オーダー 135, 176, 178, 200, 672
定義 135
～の変更 401, 560, 563
zoom
TWindow のメンバ関数 577
zoomRect
TWindow 573

【あ】

アセンブラオフセット
～の作成 660
後処理 →フェーズ
アプリケーション 28, 114, 330, 471
main() 例 32
run メンバ関数 175
例 34
TProgram のデータメンバ 472
アイドル時間 476
イベント
処理 475
イベントと 474
オーバーレイと 496
グループとしての 114
コマンドコード 305
コンストラクタ
例 32
実行 478
実行の追跡 28
実行の流れ 31
ステータスライン 477
デスクトップ 476
デストラクタ 34
デフォルト動作 39
～のカスタマイズ 260
～の起動 478
～の形態 647
～の作成 331, 473
～の終了 653
～の消去 331, 474
～の設計 276
～の動作 475
～の理解 28
パレット 475, 480
デフォルト 472
ビューとしての 133
メイン 14
メインブロック 39
メニューバー 476
メモリ 477
モードつきビューとしての 147

アプリケーションを Turbo Vision に移植
　　する　275
アンダーライン属性　332

【い】

位置
　　ストリーム入出力可能なオブジェクト
　　　　319, 324
イベント　169
　　新しい～の定義　194
　　アプリケーション
　　　　処理　475
　　ウィンドウ
　　　　処理　575
　　キーボード
　　　　次の　559
　　　　～の追跡　173
　　　　フォーカスビューと　145, 180
　　　　～へのアクセス　190
　　コマンド　185
　　コマンドと　173
　　処理　18, 30, 127, 132, 173, 187,
　　　　567
　　定数　658
　　デバッグ　272
　　なし　173
　　～の有無　556
　　～の概念　171
　　～の共用体　385
　　～のクリア　173, 191, 552
　　～の取得
　　　　次の　557
　　　　のメンバ関数　396
　　　　例　175, 193
　　～の種類　171, 658
　　～の運ばれる経路　175, 176
　　廃棄　18, 177, 191, 396
　　ビューと　143
　　フォーカス　177, 393, 660
　　　　positionalEvents と　675
　　　　キーボード　177
　　　　コマンド　177
　　　　コマンドと　184
　　　　～の経路　177, 180
　　　　例　177
　　ブロードキャスト　178, 198, 669
　　ポジショナル　143, 176, 675
　　　　コマンドと　184
　　マウス　176, 561, 673
　　　　オブジェクト　190
　　　　処理　150
　　　　次の　386
　　　　～の取得　386, 416, 451, 552
　　　　～の種類　172
　　マスク　171, 179, 306, 547, 659, 660
　　　　デバッグと　272
　　メッセージ　173, 197, 199, 320, 669
　　　　～に対する処理　199
　　～をキューにつなぐ　478, 563
　　　　定数　658
　　　　～の作成　386
　　　　～の消去　386
イベントオブジェクト　189
イベント駆動型プログラミング　32, 51,
　　168～170
色
　　TColorDisplay のデータメンバ　359
　　TColorSelector のデータメンバ　368
　　グループ　→グループ, 色
　　ダイアログボックス　356
　　バックグラウンド
　　　　～の選択　355
　　パレット　→パレット
　　表示　→画面
　　フォアグラウンド　356
色の割り当て　→パレット
色グループ
　　コンストラクタ　362
色パレット　→パレット
色項目　→項目
　　～の移動　367
　　コンストラクタ　365
　　セレクタ
　　　　～の表示　368
　　選択された色　368
　　～の選択　366

～の動作　367
～の表示と選択　365, 367
ビュー
～の作成　368
マウスとキーイベント
処理　369
リスト　366
～の作成　366
インデックス
重複　460

【う】

ウィンドウ　114, 572
アクティブ　144, 145, 678
イベント
処理　575
インテリア
複数個の　70
例　71
エディタ　613
～の作成　613
境界　388
グループとしての　141
構成要素　20
最前面の～　150
～の検出　200
スクロールバーと　577
選択された　144
タイトル　573, 575
タイル表示　59, 152, 375, 673
エラー　375
次の　653
デフォルト
動作　53, 57
～の形態　58
転送可能な　682
～のオープン
例　53
～のカスケード表示　59, 374, 673
～のクローズ　56, 193, 575, 653
アイコン　682
～の形態　61
～のサイズ　574
最大・最小値　577
～の作成　53, 574, 579,
→ウィンドウ, ～のオープン
引数　55
～のズーム　573, 577, 682
コマンド　653
～のスクロール　68
例　68
～の廃棄　56, 574, 575
～のリサイズ　682
コマンド　653
パレット　573, 575, 578, 683
番号
コマンド　654
番号づけ　55, 573, 682
1つ手前の　653
フラグ　573, 681
フレーム　58, 115, 141, 388, 573
アクティブ　145
～の作成　576
～への書き込み　62
編集用～　606
～の作成　606
～を無効にする　576
～を有効にする　576

【え】

エディタ
～の作成　582, 591
～の消去　592
ファイル　609
～の作成　609
メモリ　615
～の作成　615
～用ウィンドウ　606, 613
～の作成　606, 613
エラー
～からの回復　203
コレクション　235, 455
システム　539
処理　203
グループと　405
ストリームと　328, 329

～の検出　205
廃棄イベント　18，177，396
ファイル　208
メモリ　204，207
エラーハンドラ
システム　539
～のインストール　539，540
～の除去　539，540
演算子
ストリーム入出力可能なクラスと　322
～のチェイン　241
ビット演算　279，281

【お】

オーナービュー　56，139，140，548
基本クラスのビュー対　139
オーバーレイ
.EXE への追加　496
オーバーロード演算子
＋＝　468
void ＊()　329
等価(＝＝)　489
否定(!)　329
非等価(!＝)　489
オブジェクト
仲介～　196
～の書き込み
演算子　323
～の消去　462
～のセーブ　237
～の読み出し
演算子　323
ミュート　18

【か】

カーソル
TView のデータメンバ　546
位置　546，565
TEditor　587
入力行　418
画面からの消去　559
種類　551，562，678
起動時の　500
現在の　499
表示される　566，678
階層　98
階層図　307
影
ビュー　678
カスケード表示にしたウィンドウ
→ウィンドウ，カスケード表示にする
型
pstream のデータメンバ　327
登録済み
～へのポインタ　327
画面
色
コンストラクタ　359
～上のゴミ　64
～のクリア　501
～の初期化　501
～の高さ　499
モード
現在の　499
カラー表示
～の消去　359
環境
～の保存　255

【き】

キー　→イベント，フォーカス
ESC ～
～の検出　379
TEditor　590
TResourceItem のデータメンバ　261
イベント
色項目
処理　369
ダイアログボックス
処理　378
データ　311
～の獲得　320
局所化　90
コード
ヘッダファイル　297
コントロール～

～の変換　656
シフト～の状態
定数　297, 664
スキャンコード　660, 661
リソースと　258, 497
割り当て
TEditor　585
キーボードイベント　→イベント，フォーカス/キー
次の　559
～の追跡　173
フォーカスビューと　145, 180
～へのアクセス　190
既存のプログラム
～の移植　275
基本クラスのビュー
オーナービュー対　139
規約
書体の使用法　7
命名法　287
行
画面への書き込み　568

【く】

矩形　487
～の移動　489
空の　489
隅　488
～内の点　488
～のサイズ
～の変更　489
～の作成　488
～の比較　489
～の交わり　489
クラス
TObject からの導出～
～の消去　462
階層　97, 139
～中の基本クラス　111
ビューツリー対　138, 140
基本　462
基本的な　110
サフィクス
～のストリームへの書き込み　325
シード　102, 111
ストリーム入出力可能な　→ストリーム入出力可能なクラス
名前の付け方　287
～のインスタンス化　104
～の導出　105
表示可能な　→ビュー
プレフィクス
～のストリームへの書き込み　325
リファレンスの参照方法　310
クラスのインスタンス化　104
クラスタ　81, 116, 346,
→ラジオボタン/チェックボックス
現在選択されている項目　347
～内の項目
～に選択バーを移動する　350
～内の項目のリスト　347
～の作成　81, 347
～の消去　348
～の動作　349
～の表示　348
ヘルプコンテキスト　349
ボタンと　346
繰り返し処理メンバ関数　220, 455, 456
firstThat　221
forEach　220
lastThat　221
グループと　397, 398
コレクション と　122, 215, 220, 457
例　221, 222
クリップ　138, 557
グループ　391
TColorDialog のデータメンバ　356
TColorGroupList のデータメンバ　362
TView と　112
値
～の設定　404
～の読み出し　399
イベントと　396, 400
色
コンストラクタ　361

スクロール可能な　362
エラー処理　405
キャッシュバッファと　673
繰り返し処理メンバ関数と　397，398
サブビュー
最後の　393
～のリンク　134
サブビューの挿入　401
状態　393
定義　17
～としてのアプリケーション　140
～としてのウィンドウ　141
～の形態　137，402，405
～の再表示　402
～の作成　394
～の消去　394
～のデータサイズ　395
～の動作　400
～の表示　393，395
～のリサイズ　395
～のロック　401
バッファ　392
ビュー　56
複合　133
表示領域　392
ヘルプコンテキスト　399

【け】

継承　15，286
例　29，108
桁
TEditor　590
検索
TEditor
～用フラグ　589
置換文字列　590，668
文字列
TEditor　589
長さ　667

【こ】

項目
TColorGroup のデータメンバ　361
TColorItemList のデータメンバ　366
TStatusDef のデータメンバ　519
TStatusLine のデータメンバ　524
色　→色項目
このマニュアルで使用する書体　7
コピープロテクション　4
コマンド　184
TButton のデータメンバ　338
TEditor　586
新しい～を生成する　653
アプリケーション
コード　305
アプリケーションを終了させる　653
イベントと　173
ウィンドウのズーム　653
ウィンドウのリサイズ　653
ウィンドウ番号　655
クローズウィンドウ　652
サポートされていない　652
スクロールバー　654
～セット
処理　369
～の作成　370
～の設定　564
～の取り出し　557
選択された要素　655
ダイアログボックス　76
標準　78，653
次のウィンドウ　652
入力行　654
～の衝突　272
～の定義　47，184
～のリンク　185
ビュー　546
の有効性　653
標準　47，651，652
ダイアログボックス　78，653
フォーカスイベントと　184
フォーカスビュー　654
変更された　654
ポジショナルイベント　184
ボタンと　78，338
前のウィンドウ　653

未実装　652
呼び出し側のコマンドセット　370
～からの削除　370
～への追加　370
呼び出し側のコマンドセットからの削除　370
呼び出し側のコマンドセットへの追加　370
予約　185，651
～を無効にする　46，185，186，553
～を有効にする　186，552，555
コレクション　122，213，352，451
limit を 0 にする　470，485，531
インデックス　650
重複　460
～の作成　218
エラー　235，455
階層　214
繰り返し処理メンバ関数　220，455，456，457
最大個数　235
集合対　352
ストリーム入出力可能でない　469，484
ソート済み　122，224，459，514
キー　224，225
ダイアログボックス　354
デストラクタ　219
動的なサイズ決定　216
～内の要素のカウント　650
～の検索　650
～の作成　353，453
ストリーム入出力可能でない　469，484
～の消去　453
～の初期化　460，514
～のテスト　650
～の引数リスト　649
配列対　213，352
パック　458
ヒープと　236
ポインタと　235
メモリと　235
文字列　123，227，532
要素
～の消去　533
～の比較　533
要素　453
インデックスつき　454，457，650
コンストラクタ　217
数　452，650
全～の消去　456
ソート済み
～の検出　461
～の挿入　461
～の比較　225，460，515
～の入れ換え　454
～の検索　459
～の最大数　667
～の削除　458
～の消去　453，454，456
～の挿入　218，454，457
～の定義　217
要素数　218，452
最大　453，458
～の拡張　218，452
リストボックスと　428，430
リスト対　352
リソース　123，490
リソースと　261
例　217～219，224～226
コンストラクタ　→個々のクラスの項
コントロール
→ダイアログボックス，コントロール
値
～の設定　85
クラスタ　→クラスタ
固定文字列　→テキスト，静的
ダイアログボックスと　77，115
デフォルト　26
入力行　→入力行
ヒストリーリスト　→ヒストリーリスト
フェーズと　181
フォーカス　80
インジケータ　549
画面上での形態　145
デフォルト　81

〜へのラベルのリンク　83，424
ボタン　→ボタン
ラベル　→ラベル
リストビュアー　→リストビュアー
リストボックス　→リストボックス

【さ】

座標　128，130，467，487
グローバル　560
ローカル　560
サフィクス
ストリームへの書き込み　325
サブビュー　56，113，126，133，140，
→ビュー
イベントと　400
繰り返し処理メンバ関数と　397，398
グループへのアタッチ　134
最後の　393
最初の　397，560
選択された　393，403，564
次の　562
〜の順序　560，562，563
〜の消去　402
〜の挿入　401
〜の定義　17
〜の廃棄　143
〜の表示　395
1つ手前の　562
フォーカス　→ビュー，フォーカス
〜へのポインタ　395

【し】

識別子
Uses_TClassName　301
システムエラーハンドラ　→エラーハンドラ，システム
持続的なオブジェクト　213，237〜255，
→ストリーム入出力可能なクラス
集合
コレクション対　352
ショートカットキー　→ホットキー
初期化　→個々のクラスのコンストラクタ
信頼性の高いプログラミング　203
例　211

【す】

スキャンコード　311
キー　661
キーボード　660
スクローラ　118，508
delta 値　70，508
最大値　509
〜の設定　511
コンストラクタ　70
スクロールバーと　505，509
〜の形態　511
〜のサイズ
〜の変更　510
〜の作成　509
〜の動作　511
パレット　510，512
スクロールバー　119，502
TEditor　589，591
値　503，506
最小　503
〜の設定　506
最大　502
〜の設定　506
〜の設定　506，507
コンストラクタ　504
最大値
TEditor　590
スクローラと　505，509
〜の値の変化
コマンド　655
〜のクリック
コマンド　655
〜の形態　502，505
〜の動作　505
〜の部品　676，680
コード　305
パレット　505，507
標準　577
フェーズと　180
部品　506
ページング　503

矢印　502
リストビュアーと　432, 433
スコープ
モードつきビューと　147
ステータス項目
～の作成　45, 522
ステータス定義
～の作成　45, 520
ステータスライン　120, 523
items　524
現在の　473
コマンド
～の発行　184
～のリンク　45, 185
定義　523
～の形態　525
～の更新　526
～の作成　44, 45, 471, 473, 477, 524, 674
例　47
～の消去　474
～の使用法　20
～の初期化
例　45
～の動作　525
～の廃棄　524
パレット　525, 527
ヒント文字列　526
ヘルプコンテキストと　201, 523
ポジショナルイベントと　184
ホットキーのリンク　46
モードつきビューと　147
ストリーム　121, 237～255
TResourceFile のデータメンバ　494
～からの読み出し　247
文字列の　533
要素の　533
状態　327
登録　121, 262, 301, 529
例　254
～の終わり　328
～の初期化　329
～の定義　239
～のフラッシュ　324
～の読み書き
エラー　328, 329
バッファ
～へのポインタ　327
バッファを使用する　→バッファ
ファイル
双方向の　314
～への書き込み　247, 325, 326
文字列の　534
要素の　534
～へのデスクトップの格納　255
リソースと　261
ストリームマネージャ
TStreamable と　245
TStreamableClass と　245
TStreamableTypes と　245
コード
～へのリンク　251
ストリーム入出力可能なオブジェクト
基本的な操作　312
～内の位置　319, 324
～の組み立て　247
～の検出　318, 324
～のフラッシュ　324
～の読み書き
ifpstream　315
ipstream　318
ofpstream　321
演算子　323
現在位置　318
双方向の　314
同時に　316
ストリーム入出力可能なクラス　527,
→持続的なオブジェクト
build メンバ関数　247
BUILDER typedef と　530
pstream と　238
read メンバ関数　247
TGroup と　238
TView と　238
write メンバ関数　247
エディタ　582

演算子と　322
コンストラクタ　248
データベース　469, 484
～の作成　531
登録　251, 302, 528,
→ストリーム, 登録
登録済み
データベース　531
名前の付け方　250, 528
～の作成　237, 251, 527, 530
～の定義　244
～の読み書き　323
基本クラスの　317, 327
文字列の　318
ポインタと　469
メモリオーバーヘッド　245
スペースキー
ダイアログボックスと　80

【せ】

制御文字列　→文字列, 制御
静的
テキスト　→テキスト, 静的
メンバ　107, →データメンバ/メンバ関数
セーフティプール　204
delete 演算子と　674
lowMemory 関数と　204
new 演算子と　205, 674
TBufListEntry と　205
TVMemMgr と　205
validView と　207
エラーチェックと　206
従来のエラーチェック対　205
大量消費ビューと　210
デフォルトサイズ　204
～のサイズ　204, 570, 656
～の割り当て　674
メモリ割り当て　570
例　206
選択バー
クラスタ項目への移動　350
センタリング　→ビュー, センタリング

【そ】

挿入点　→入力行, カーソル
添え字演算子([])　464
ソート済みリストボックス　645
～の作成　645
属性
バックグラウンド　368
表示バッファ　384
フォアグラウンド　368
ワード
ロングワードからの取り出し　293
ソフトウェアライセンスの同意書　4

【た】

ターミナル　119, →テキスト
TTY　543
～の作成　544
行　542
オフセット　542
スクローラ
～の再表示　542
ダム　540
～の作成　541
～の消去　541
～の形態　542
バッファ　540
オーバーフロー　544
オフセット　542
キュー　542
キューのオフセット　541
～のサイズ　540
～への挿入　542
パレット　543, 544
ダイアログボックス　115, 377, 637
TAB キーと　80
値
～の格納　89
～の設定　85, 86
例　88
～の読み出し　86
色　358
～の作成　356

色グループ　356
ウィンドウ対　75
キーイベント
　ESC キー　379
　処理　378
境界　388
固定文字列　→テキスト，静的
コマンド
　No ボタン　653
　OK ボタン　653
　Yes ボタン　653
　キャンセル　653
　デフォルトボタン　653
　標準　78，653
コレクション　354
コントロール　77，92
　値
　　～の設定　85
　ホットキー　89
スペースキーと　80
チェックボックス　→チェックボックス
ディレクトリ　621，625
ディレクトリ変更用の　617
デフォルト動作　76
入力行　→入力行
～のオープン　74
　例　74，205，206
の概要　31
～のキャンセル　76
～のクローズ　76
～の作成　377，637
～の使用　25
～の設計　77
～の動作　357
～の表示
　TEditor　588
パレット　378，379，578
　現在の　356，357
ヒストリー選択リスト　632
　～の作成　634
　～の消去　635
ヒストリーリスト　→ヒストリーリスト
標準　94
ファイル　632
　～の作成　634
　～の消去　635
ファイルのオープン　94
ファイルコレクション　628
　～の作成　628
ファイル名　639，641
　～の作成　639，642
フレーム　388
ボタン　→ボタン
ホットキー　89
　衝突　89
モードつき　76
　例　76
モードなし　76，77
　例　74
ラジオボタン　→ラジオボタン
ラベル　→ラベル
リストビュアー　→リストビュアー
リストボックス　→リストボックス
リターンキーと　80
タイトル
　入力ボックス　663，664
　ボタン　339
　文字列
　　ウィンドウ　573，575
代入
　TRect メンバ関数　130
代入演算子　464
大量消費ビュー　210
タイル表示にしたウィンドウ
　→ウィンドウ，タイル表示
対話型プログラム　22～34
　エラー処理　203
　原則　24，27
多国語化　267
　リソースと　260
多態性　106
ダム端末　→ TTerminal
単位的動作　204
　valid メンバ関数と　210
　セーフティプールと　204

【ち】

チェックボックス　116，343
　TLabel オブジェクトと　343
　値　82，345
　　～の設定　345
　説明　81
　～に値のセット　82
　～の作成　81
　～の消去　344
　～の反転　345
　～の表示　345
　パレット　346
　マークされた　345
　例　81
置換文字列
　長さ
　　最大　668
抽象
　クラス　→クラス，抽象
　メンバ関数　287
重複
　TNSSortedCollection のデータメンバ
　　227，460
　　～の使用　258

【て】

定数　→個々の定数型
　TEditor　585
　アプリケーションパレット　647
　コマンド　651
　プレフィクス　287
　ボタンフラグ　649
ディスク
　マスター
　　～の内容　4
ディスプレイ　→画面
ディスプレイアクセス　18
ディレクトリ
　ダイアログボックス　617，621，624
　　～の作成　618，621
　～の表示　626
　パスとテキスト　625
　～用のリストボックス　625
ディレクトリ変更
　～用のダイアログボックス　617
データメンバ　108，
　→個々のデータメンバ名
　静的　107
　名前の付け方　287
　リファレンスの参照方法　310
テキスト
　書式化済み　464
　ヒストリーリスト　411
　表示バッファ　383
テキストデバイス　→ターミナル
デスクトップ　113，373
　現在の　472
　～上でのウィンドウのカスケード表示
　　59，374
　～上でのウィンドウのタイル表示
　　59，152，375
　　エラー　375
　ストリームと　255
　ストリームにセーブする　255
　～の形態　375
　～の作成　374，471，473，476
　～の消去　474
　～の動作　375
　バックグラウンド　138
デストラクタ　→個々のクラスの項
デバッグ　271
　イベント処理　272
　コマンド　272
点　467

【と】

動作
　単位的　204
等値演算子(==)　489
登録，ストリーム　→ストリーム，登録

【な】

名前の付け方　287

【に】

入力ボックス
　～の表示　663，664
入力行　84，117，417
　値
　　～の設定　420，422
　カーソル
　　位置　418
　コマンド　654
　コンストラクタ　84，419
　選択された　419，421
　データ　418
　　～のサイズ　420
　動作　84，421
　長さ
　　最大　418
　～の消去　420，642
　～の表示　420
　パレット　421，423
　ヒストリーリストと　407
　フェーズと　182
　例　84

【の】

ノーマル属性　332

【は】

バイト
　ストリームに書き込む　325
ハイライト属性　332
配列
　コレクション対　352
バックグラウンド　334
　TDeskTop のデータメンバ　373
　オブジェクト
　　～の作成　372
　デスクトップ　138
　～の形態　335，336，375
　～の作成　334，335
　パターン　334
　パレット　335
バッファ
　modified フラグ
　　TEditor　590
　キャッシュ　673
　グループ　392，673
　デフォルト
　　～の割り当て　328
　～の解放　541
　ファイル
　　現在の　313
　　～の作成
　　　fpbase　312
　　　ifpstream　315
　　　iopstream　317
　　　ipstream　317
　　　ofpstream　321
　　　opstream　324
　　　pstream　328
　　　双方向の　314
　　～の割り当て　313
バッファを使用するビュー
　→ビュー，バッファ
パラメータつきのテキスト文字列　464
パレット　160～165，463
　cpBackground　336
　cpBlueWindow　578
　cpButton　342
　cpCluster　346
　cpCLuster　351
　cpCluster　487
　cpCyanWindow　578
　cpDialog　379
　cpFrame　390
　cpGrayWindow　578
　cpHistory　410
　cpHistoryViewer　412
　cpHistoryWindow　414
　cpInputLine　423
　cpLabel　426
　cpListViewer　431，438
　cpMenuView　440，443，448
　cpScrollBar　507
　cpScroller　512，543，544
　cpStaticText　467，518

cpStatusLine 527
getColor と 162, 557
TWindow 305
アプリケーション 475, 480, 647
ウィンドウ 575, 578, 683
クラスタ 349
固定文字列 518
コレクション 354
スクローラ 510, 512
スクロールバー 505, 507
ステータスライン 525, 527
ダイアログボックス 378, 379, 578
　現在の 356
端末 543
チェックボックス 346
テキストデバイス 544
デフォルト～ 160
　～のオーバーライド 163
入力行 421, 423
ヌル 161
～の拡張 164
～の作成 464
～の消去 464
バックグラウンド文字列 335
パラメータつきのテキスト文字列 467
ヒストリーウィンドウ 414
ヒストリーリスト 408, 410
ヒストリーリストウィンドウ 414
ヒストリーリストビュアー 411, 412
ビュー 558
フレーム 389, 390
ボタン 340, 342
マッピング 161
無効な選択 547
メッセージ 578
メニュー 446, 448
メニューバー 440
メニューボックス 443
ラジオボタン 487
ラベル 425, 426
リストビュアー 435, 438
リストボックス 431
レイアウト 160
ハングアップ（プログラムの）
　デバッグ 274

【ひ】

ヒープ
　コレクションと 236
　セーフティプール 204
ヒストリー選択リスト
　～の作成 634
　～の消去 635
　～の用意 632
ヒストリーリスト 93, 117, 407
　ID 番号 411
　アイコン 408
　インデックス 663
　ウィンドウ 413
　　コンストラクタ 413
　　パレット 414
　　ビュアーと 413
　コンストラクタ 408
　入力行と 407
　～の形態 408
　パレット 408, 410
　ビュアー 410
　　ウィンドウと 413
　　コンストラクタ 411
　　テキスト 411
　　～の形態 412
　　～のサイズ 412
　　～の作成 406
　　～の挿入 406
　　～の動作 412
　　パレット 411
　～への文字列の追加 663
　文字列
　　の数 663
ビット
　～のクリア 280
　～のセット 279
　～のテスト 280
　～の反転 280
　～のマスク 281
ビットマップフィールド 278, 279, 281

否定演算子(!)
　～のオーバーロード　329
ビデオ
　リバース
　　モノクロ　332
非等値演算子(!=)　489
ビュー　112, 126, →サブビュー
　draw メンバ関数 と　127
　exposed　556
　state フラグ　549
　valid　567
　　TEditor　590
　位置
　　～の設定　564
　イベントと　127, 132, 187, 559, 567
　色選択用の　367
　インテリア　58
　　フレームつき　59
　　例　59
　エラー処理　567
　オーナー　→オーナービュー
　オーバーラップ　134
　オプションフラグ　548, 672
　階層　545
　影のついた　678
　画面からの消去と除去　550, 559
　～間の通信　196, 669
　～間のメッセージ　197
　グローモード　547, 661
　コマンド
　　現在のセット　546
　　コード　303
　最前面の
　　～の検出　200
　再表示　590
　垂直方向のセンタリング　153
　水平方向のセンタリング　153
　ストリームと　244
　選択された　144, 393, 403, 564, 678
　選択可能な　150, 672
　センタリング　153, 673
　挿入　56, 401
　ターミナル　115, 133
　　イベントと　177
　ツリー　→ビューツリー
　定義　17, 126
　データ
　　～のサイズ　552
　　～の設定　565
　　～の読み出し　557
　テキスト　358, 359
　　色　359, 360
　　～の作成　359
　　～の動作　359
　デバッグ　273
　動作　187, 559
　　デフォルト　169
　　～の変更　149
　～としてのアプリケーション　126, 133
　ドラッグ　553, 678
　ドラッグモード　546
　～の位置　110, 128, 548, 556, 558
　　～の変更　551, 560, 562
　～のグループ　56, 113, 133
　～の形態　112, 127, 131
　～の検出　198
　～のサイズ　110, 128, 129, 549
　　最大・最小値　567
　　最大値　668
　　～の変更　551, 558
　～の作成　550
　～の表示　61, 553, 555
　　lock メンバ関数と　401
　　unlock メンバ関数と　405
　　テキストと色　359
　　バッファ　64
　　バッファを使用する
　　　例　65
　バッファを使用する　151
　幅
　　最大値　668
　パレット　160～165, 480, 557, 558
　表示　566
　表示可能な　566, 678
　フォーカス　18, 144～146, 678
　　イベントと　177

コマンド　654
デフォルト　145
モノクロ　680
フレームつき　151, 672
ヘルプコンテキスト　548, 558
マウスイベントと　552
無効　678
モードつき　→モードつき, ビュー
有効　678
有効性のチェック　479, 653
リサイズ　153
ビューツリー　56, 139, 140
クラス階層対　138, 139
～の構築　140
～の切除　143
プログラムの制御フローと　143
表記上の約束　7
表示バッファ　64
画面への書き込み　568
属性を設定する　384
パレットと　67
～へのテキストの転送　383
文字の転送　383
文字を設定する　384
文字列の転送　383
～用のクラス　382
ビルダ（組み立て関数）　247
ヒント文字列
ステータスラインと　526

【ふ】

ファイル
HELPME!.DOC　5, 6
I/O　→I/O
README.DOC　5
signature　495
エディタ　609
～の作成　609
コレクション
ダイアログボックス　628
～の作成　628
サイズ
TEditor　590
ストリーム
双方向の　314
ダイアログボックス　632
～の作成　634
～の消去　635
～のアタッチ　313
fpbase　312
ifpstream　315
ofpstream　321
opstream　324
双方向の　314
～のオープン　312
書き込み用として　321, 324
双方向の　314
モード　313, 315, 316, 322
読み出し用として　315
～のクローズ　313
バッファ
現在の　313
～の割り当て　313
モード
～の 設 定　312～316, 321, 322, 324
～用の入力行　632
ダイアログボックス　639
～の作成　639
～の作成　634
～の消去　635
リスト
ダイアログボックス　641
～の作成　642
リソース　→リソース
ファイルストリーム
～の作成　258
フェーズ　393
TGroup のデータメンバ　181, 393
後処理　151, 180, 673
前処理　151, 180, 673
フォーカス
TListViewer のデータメンバ　432
イベント　→イベント, フォーカス
コントロール
→コントロール, フォーカス

ビュー　→ビュー，フォーカス
要素　→要素，フォーカス
フォーカスチェイン　→ビュー，フォーカス
イベントと　177
フォーマット状態フラグ　327
複数個のインテリア　70
フラグ　149，278，281
options　150
state　149，549，677
TButton のデータメンバ　338
TWindow のデータメンバ　573
マスク　305
ウィンドウ　573，681
エディタの状態　591
オプション　278，548，672
検索　589
～のクリア　280
～の設定　279
～の定義　278
～のテスト　280
～の反転　280
～の読み方　278
ボタン　338，649
ブレークポイント　271
handleEvent　272
ビュー　273
プログラムのハングアップと　274
フレーム　388
TWindow のデータメンバ　573
ウィンドウ　58，115，141，573
アクティブ　145
～の作成　576
～の作成　388，579
～の表示　389
パレット　389，390
ビュー　151，672
マウスイベント
処理　389
プレフィクス
～のストリームへの書き込み　325
ブロードキャストイベント
→イベント，ブロードキャスト
ブロック選択
TEditor　584

【へ】

ヘッダファイル
TV.H によるインクルードの制御　301
条件インクルード　42
～の作成　302
ヘルプコンテキスト
ステータスラインと　202，523
定数　305，662
～の取り出し
クラスタ　349
グループ　399
ビュー　558
メニュー　446
ビューと　548
フォーカスビューと　201
メニュー項目　49
予約済みの　662

【ほ】

ポインタ
0（ゼロ）　266
void への～
～のオーバーロード　329
サブビューへの～　395
ストリームバッファ　327
登録してある型　327
バッファ
pstream　329
リンクリスト　361，364
ポジショナル イベント
→イベント，ポジショナル
ボタン　21，26，78，115，337
Cancel　26，79
OK　80
押される　341
クラスタと　346
コマンド　78，338
～のリンク　78
状態　342
タイトル　339
デフォルト　26，80，649

コマンド　653
～の作成　341
～の数　338
～のチェック　338
ノーマル　338, 649
～の作成　78, 339
～の消去　340
～の初期化　338
～の動作　182, 340
～の表示　340
パレット　340, 342
フェーズと　182
フラグ　338, 649
マウス　668
ラベル　78, 339, 649
左詰め　338
例　78
～を有効/無効にする　338
ボックス
入力　664
～の表示　663
メッセージ　669, 670
ホットキー
衝突　89
ダイアログボックス　89
～の局所化　90, 90
フェーズと　182
メニューと　447

【ま】

マウス
イベント　561, 673
色項目
処理　369
オブジェクト　190
種類　172
処理　150
次の　386
データ構造体　451
～の取得　386, 416
フレームと　389
ポジショナル　176
呼び出したビューの内側の～　552
カーソル
～の画面からの消去　416, 449
～の表示　417, 449, 450
～処理の中止　417, 450
～の位置　561
～の復旧　416, 450
ハンドラ
～の設定　416
～の登録　450
ボタン　668
～の設定　449
有効　416, 450
有効範囲
～の設定　416, 450
前処理　→フェーズ
マクロ
__link　263
マスク　278
イベント　547, 660
ビットマップフィールドと　281
マスターディスク
～のバックアップ　4
マスターディスク, 内容　4

【み】

ミュートオブジェクト　18

【め】

メッセージ
TScrollBar　306
イベント　173
パレット　578
標準　306
ボックス　669, 670
メニュー　116, 438, 443,
→メニューバー/メニューボックス
TMenuView のデータメンバ　444
～間のリンク　444
構成要素　20
項目　444, 446
ショートカット　445
選択された　444
ショートカットと　49, 445

ネストした　674
～の作成　444
～の操作　24
～の動作　446
パレット　446, 448
ヘルプコンテキスト　446
ホットキーと　49, 447
メニューバー　438, →メニュー
現在の　472
コンストラクタ　439
例　50
～の形態　439
～の作成　471, 473, 476
～の消去　474
パレット　440
ヘルプコンテキストと　49
マウスと　439
例　49
メニューボックス　441, →メニュー
コンストラクタ　441
～の形態　442
パレット　443
マウスと　442
メモ
エディタ　615
～の作成　615
メモリ
アプリケーションと　477
エラー　203～205, 235
コレクションと　235
ストリーム入出力可能なクラスと　245
セーフティプール　204, 667
～の解放　536, 538
バッファ　541
文字列リスト　536, 538
～の大量消費ビュー　210
マネージャ　667
割り当て　204
グループバッファと　673
セーフティプール　570
メンバ
静的　107
メンバアクセス
→データメンバ/メンバ関数
リファレンスの形式　310
メンバ関数
仮想　15, 106, 287
仮想でない　107
空の　103, 106
繰り返し処理関数
→繰り返し処理メンバ関数
静的　107
抽象　287
デフォルト　106
名前の付け方　287
～のオーバーライド　105, 107
リファレンスの参照方法　311

【も】

モードつき
ダイアログボックス　76, 147
～の終了　78
ビュー　147, 679, →ビュー
イベントと　176～178
現在の　567
スコープと　147
ステータスラインと　147
～としてのアプリケーション　147
～の実行　397, 556
～の終了　396, 555
モードなしダイアログボックス
→ダイアログボックス, モードなし
文字/キャラクタ
インジケータ
モノクロビデオモードでのフォーカスビュー　680
コード　311, 320
表示バッファ　383, 384
ワード
ロングワードからの取り出し　293
文字列
TCluster のデータメンバ　347
画面への書き込み　569
ストリームへの書き込み　325
制御
長さ　655

テキスト
パラメータつきの　464
～の検索
長さ　667
～のコレクション　532
～の作成　532，533
～の読み出し　318
～の割り当て　671
表示バッファ　383
リスト　→文字列リスト
リンクリスト　513
文字列の検索
長さ
最大　667
文字列リスト　124，535
～からの文字列の取り出し　536
～のアクセス　535
～の作成　269，536～538
～の使用　267
～の定義　267
～への文字列の追加　538
メモリ
～の解放　536，538
リソースファイルと　267
モデム
～を介してオブジェクトを転送する　327
モノクロ
クラスタ　332，333
属性
セレクタ　332
フォーカスコントロール
インジケータ文字　549

【よ】

要素　→コレクション
TListBox のデータメンバ　428
TNSCollection のデータメンバ　216，453
コレクションと　453
選択された
コマンド　655
フォーカス
文字列値　414
リストビュアー　432，435，436
リストビュアー
番号　432
リストボックスと　428～430
予約コマンド　→コマンド，予約

【ら】

ライセンス契約　4
ライタ（書き込み関数）　246
定義　246
ライブラリ
～の作成　660
ラジオボタン　116，485
値　83
～の設定　486，487
～の読み出し　486
説明　81
～の形態　486
～の作成　81
パレット　487
例　83
ラベル　83，423
コンストラクタ
例　84
入力ボックス　663，664
～のコントロールへのリンク　83，424
～の作成　424
～の動作　425
～の表示　425
バックグラウンド　355
フォアグラウンド　356
ボタン　339

【り】

リーダ（読み込み関数）　247
～の定義　246
リストビュアー　92，118，431
サイズの変更　434
スクロールバーと　432，433
段数　432
テキスト
～のコピー　363

～のサイズ　432
～の作成　362, 406, 433
～の消去　362
～の動作　435
～の表示　434
パレット　435, 438
要素
先頭～　433
～の選択　436
～の取り出し　435
番号　432, 436
フォーカス　432, 435, 436
リストボックス　93, 118, 427
値
～の取得　429
～の設定　430
コレクションと　118, 428
コンストラクタ　428
ソート済み　645
～の作成　645
ディレクトリ表示用の～　625
データ
～のサイズ　429
～の消去　626
パレット　431
要素　428
～の入れ替え　430
～の取り出し　429
リソース　123, 213, 257～269
0 ポインタと　266
fpstream と　258
TResourceCollection と　257
TResourceFile と　257
位置とサイズに関するデータ　258
階層　214
カスタマイズと　260
コレクション　491
コンストラクタ　491
コレクションと　261, 490
ストリームと　261
～によるコードサイズの縮小　260
～の格納　493
～の作成　262
例　262
～の使用　260
～の除去　497
～の読み出し　265
例　265
ファイル　493
.EXE への追加　496
インデックス　497
ストリームと　494
～のサイズ　496
～の作成　494
～のフラッシュ　496
～の読み出し　497
～への書き込み　497
文字列リストと　267
リバースビデオ属性　332
リンク
コレクション対　352
文字列　→文字列リスト
リンク　361
色の名前とインデックスの～　364
色インデックス　364
色グループ　361
色項目　361, 364
～の作成　361

【る】

ルックアンドフィール　19

【わ】

ワード
ストリームへの書き込み　326

Turbo Vision ™ for C＋＋　ユーザーズガイド

1992年　1月　6日　初版発行

著者　**Borland International**

編訳・発行　**ボーランド株式会社**

〒107 東京都港区南青山7-8-1 小田急南青山ビル

TEL　(03)3486-1400(代表)

FAX　(03)3486-1423

Printed in Japan

落丁，乱丁はお取り替えいたします。

BORLAND® C++
BORLAND
Copyright ©1987, 1991 Borland International, Inc. All rights reserved. Licensed material—program property of Borland International, Inc. 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001. Unauthorized use, duplication or distribution is strictly prohibited by federal law.
APPLICATION FRAMEWORKS
BORLAND® C++
BORLAND
Copyright ©1987, 1991 Borland International, Inc. All rights reserved. Licensed material—program property of Borland International, Inc. 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001. Unauthorized use, duplication or distribution is strictly prohibited by federal law.
APPLICATION FRAMEWORKS
BORLAND® C++
BORLAND
Copyright ©1987, 1991 Borland International, Inc. All rights reserved. Licensed material—program property of Borland International, Inc. 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001. Unauthorized use, duplication or distribution is strictly prohibited by federal law.
APPLICATION FRAMEWORKS
BORLAND® C++
BORLAND

USER'S GUIDE

TURBO VISION™ FOR C++

BORLAND